الجمهورية الشعبية الديمقراطية الجزائرية
People's Democratic Republic of Algeria
وزارة التعليم العالي و البحث العلمي
Ministry of Higher Education and Scientific Research
المدرسة العليا للإعلام الآلي 8 ماي 1945 - سيدي بلعباس
Higher School of Computer Science
8 Mai 1945 - Sidi Bel Abbes

[Graduation/Master's] Thesis

To obtain the diploma of [**Engineering/Master's**] **Degree**

Field of Study: **Computer Science**

Specialization: [**Specialization**]

Theme

Optimizing LLM Integration Patterns in Enterprise Applications. A Comparative Study of REST, MCP, and Streaming Protocols

---

Presented by
Akermi Yahia Abderaouf

Defended on: **September, 2025**
In front of the jury composed of

Mr. [Jury Member Name]	President of the Jury
Mr. [Jury Member Name]	Thesis Supervisor
Mr. [Jury Member Name]	Co-Supervisor
Mr. [Jury Member Name]	Examiner

Academic Year: 2024/2025

Acknowledgement

Acknowledgement

I would like to express my deepest gratitude to the authors of the three seminal works that have significantly influenced the direction and depth of my research on "Optimizing LLM Integration Patterns in Enterprise Applications. A Comparative Study of REST, MCP, and Streaming Protocols."

Firstly, I am indebted to Vaswani et al. for their groundbreaking work on the Transformer architecture, as presented in "Attention Is All You Need." Their innovative approach to sequence transduction using attention mechanisms has laid the foundation for modern large language models (LLMs) and inspired the technical underpinnings of this thesis.

Secondly, I extend my heartfelt thanks to the OpenAI team for their pioneering research on GPT-3, as detailed in "Language Models are Few-Shot Learners." Their exploration of scaling LLMs and the introduction of few-shot learning capabilities have provided invaluable insights into the practical deployment and integration of LLMs in enterprise applications.

Lastly, I am grateful to the authors of the DiXtill paper for their novel contributions to knowledge distillation and explainable AI. Their work has been instrumental in shaping my understanding of efficient LLM deployment and the importance of interpretability in enterprise contexts.

This thesis would not have been possible without the inspiration and knowledge derived from these exceptional works. I am deeply appreciative of their contributions to the field of artificial intelligence and their impact on my academic journey.

I would also like to thank my advisors, colleagues, and family for their unwavering support and encouragement throughout this research endeavor.

Abstract

Abstract

Large Language Models (LLMs) have revolutionized natural language processing, enabling a wide range of applications in enterprise environments. However, the integration of LLMs into enterprise systems presents unique challenges, particularly in selecting the most efficient communication protocol for deployment. This thesis investigates and compares three integration patterns—REST APIs, Message Control Protocols (MCP), and Streaming Protocols—focusing on their performance, scalability, and suitability for enterprise applications.

The study begins by exploring the architectural foundations of LLMs, including attention mechanisms and transformer-based models, to establish a technical context. It then evaluates the three integration patterns across key metrics such as latency, throughput, resource utilization, and adaptability to real-time and batch processing scenarios. The analysis is supported by experimental results derived from deploying LLMs in simulated enterprise environments.

This thesis also examines the trade-offs between protocol simplicity, parallelization capabilities, and real-time responsiveness, providing actionable insights for enterprise decision-makers. By synthesizing these findings, the research highlights best practices for optimizing LLM integration and offers a roadmap for future developments in enterprise AI systems.

Keywords— Large Language Models, REST APIs, Message Control Protocols, Streaming Protocols, Enterprise Applications, Integration Patterns

Acronyms

Contents

List of Figures

List of Tables

Part 1

Introduction

1.1 Introduction

Large Language Models (LLMs) have emerged as transformative tools in natural language processing, enabling a wide range of applications across industries. From customer support chatbots to advanced decision-making systems, LLMs are increasingly integrated into enterprise applications to enhance efficiency, scalability, and user experience. However, the integration of LLMs into enterprise systems presents unique challenges, particularly in selecting the most suitable communication protocol for deployment.

Enterprise applications often rely on communication protocols to facilitate interactions between clients and servers. Among these, Representational State Transfer (REST), Model Context Protocol (MCP), and Streaming protocols have gained prominence due to their distinct characteristics and use cases. REST APIs are widely adopted for their simplicity and stateless nature, making them ideal for traditional request-response interactions. MCP, on the other hand, offers message-based communication, enabling asynchronous and context-aware interactions. Streaming protocols, such as WebSocket, provide real-time, bidirectional communication, making them suitable for applications requiring low-latency responses.

The choice of communication protocol significantly impacts the performance, scalability, and user experience of LLM-powered applications. Factors such as latency, throughput, resource utilization, and adaptability to real-time or batch processing scenarios must be carefully considered. Despite the growing adoption of LLMs, there is a lack of comprehensive studies comparing these protocols in the context of enterprise applications.

This thesis aims to address this gap by conducting a comparative study of REST, MCP, and Streaming protocols for integrating LLMs into enterprise systems. By evaluating these protocols across key metrics and use cases, this research seeks to provide actionable insights for enterprise decision-makers. Additionally, the thesis explores the architectural foundations of LLMs, including attention mechanisms and transformer-based models, to establish a technical context for the study.

The findings of this research are expected to contribute to the optimization of LLM integration patterns, enabling enterprises to leverage the full potential of LLMs while addressing the challenges associated with protocol selection. This work not only highlights the trade-offs between protocol simplicity, parallelization capabilities, and real-time responsiveness but also offers a roadmap for future developments in enterprise AI systems.

1.2 Challenges

The integration of Large Language Models (LLMs) into enterprise applications introduces several challenges that must be addressed to ensure efficient and reliable deployment. One of the primary challenges is the selection of an appropriate communication protocol. Each protocol—REST, Model Context Protocol (MCP), and Streaming—has its own strengths and limitations, and the choice can significantly impact performance, scalability, and user experience.

REST APIs, while simple and widely adopted, may struggle with high-latency scenarios and lack support for real-time interactions. MCP, with its message-based architecture, offers better support for asynchronous communication but introduces complexity in managing message queues and ensuring context consistency. Streaming protocols, such as WebSocket, excel in real-time, low-latency applications but require robust infrastructure to handle continuous bidirectional communication.

Another challenge lies in optimizing resource utilization. LLMs are computationally intensive, and their integration can strain enterprise infrastructure, leading to increased costs and potential bottlenecks. Balancing resource allocation while maintaining high throughput and low latency is a critical concern. Additionally, ensuring the security and privacy of data exchanged between clients and servers is paramount, especially in industries like finance and healthcare where sensitive information is involved.

1.3 Motivation

The rapid adoption of LLMs in enterprise applications underscores the need for optimized integration patterns that can unlock their full potential. Enterprises rely on LLMs for tasks ranging from customer support and content generation to decision-making and predictive analytics. However, the lack of a clear understanding of how different communication protocols impact LLM performance and scalability creates a significant gap in the field.

This thesis is motivated by the need to provide actionable insights into the trade-offs associated with REST, MCP, and Streaming protocols. By systematically evaluating these protocols, this research aims to empower enterprise decision-makers to make informed choices that align with their specific use cases and operational requirements. Furthermore, the study seeks to address the challenges of resource optimization, latency reduction, and real-time responsiveness, which are critical for the successful deployment of LLMs in enterprise environments.

1.4 Organisation and Structure

This thesis is organized into five main parts, each addressing a specific aspect of the research:

- **Part 1: Introduction**
  This part introduces the research topic, outlines the challenges and motivation, and presents the objectives and scope of the study.
- **Part 2: Background**
  This part provides the theoretical foundation for the study, covering the architecture of LLMs, the principles of REST, MCP, and Streaming protocols, and a review of related work.
- **Part 3: Implementation**
  This part details the design and development of a Spring Boot application that integrates a distilled LLM using REST, MCP, and Streaming protocols. It also describes the metrics collection and monitoring setup.
- **Part 4: Results**
  This part presents the experimental setup, results, and analysis, highlighting the trade-offs and performance metrics for each protocol.
- **Part 5: Conclusion**
  This part summarizes the key findings, discusses the limitations of the study, and suggests directions for future research.

The document concludes with a comprehensive bibliography, listing all references cited throughout the thesis.

Part 2

Background

2.1 Background

2.1.1 Introduction to Large Language Models (LLMs)

Large Language Models (LLMs) have fundamentally transformed the landscape of natural language processing (NLP) and artificial intelligence applications across industries. These sophisticated neural network architectures, built upon the foundation of deep learning principles, have demonstrated unprecedented capabilities in understanding, generating, and manipulating human language at scale. The evolution from early statistical language models to contemporary transformer-based architectures represents one of the most significant breakthroughs in computational linguistics and machine learning.

LLMs are characterized by their massive parameter counts, often ranging from hundreds of millions to hundreds of billions of parameters, enabling them to capture intricate patterns and relationships within textual data. This scale allows them to perform diverse linguistic tasks without task-specific architectural modifications, a property known as task-agnostic performance. The versatility of LLMs has made them indispensable tools in enterprise applications, ranging from customer service automation to content generation and decision support systems.

The fundamental capabilities of modern LLMs include:

- **Multi-task Performance:** Executing a comprehensive range of NLP tasks including text generation, summarization, translation, question answering, sentiment analysis, and code generation without requiring task-specific fine-tuning.
- **Few-shot Learning:** Adapting to new tasks with minimal examples, leveraging pre-trained knowledge to understand context and requirements from just a few demonstrations.
- **Contextual Understanding:** Processing and maintaining coherent understanding across long sequences of text, enabling complex reasoning and discourse comprehension.
- **Emergent Abilities:** Displaying capabilities that were not explicitly programmed but emerge from the scale and complexity of the training process.
- **Scalable Architecture:** Benefiting from increased computational resources and data, following predictable scaling laws that relate model size to performance improvements.

The enterprise adoption of LLMs has been driven by their ability to automate complex language-based tasks that previously required human expertise. Organizations across sectors including finance, healthcare, legal services, and technology have integrated LLMs into their workflows to enhance productivity, reduce operational costs, and improve service quality. However, this integration presents unique challenges related to computational efficiency, latency requirements, security considerations, and system interoperability.

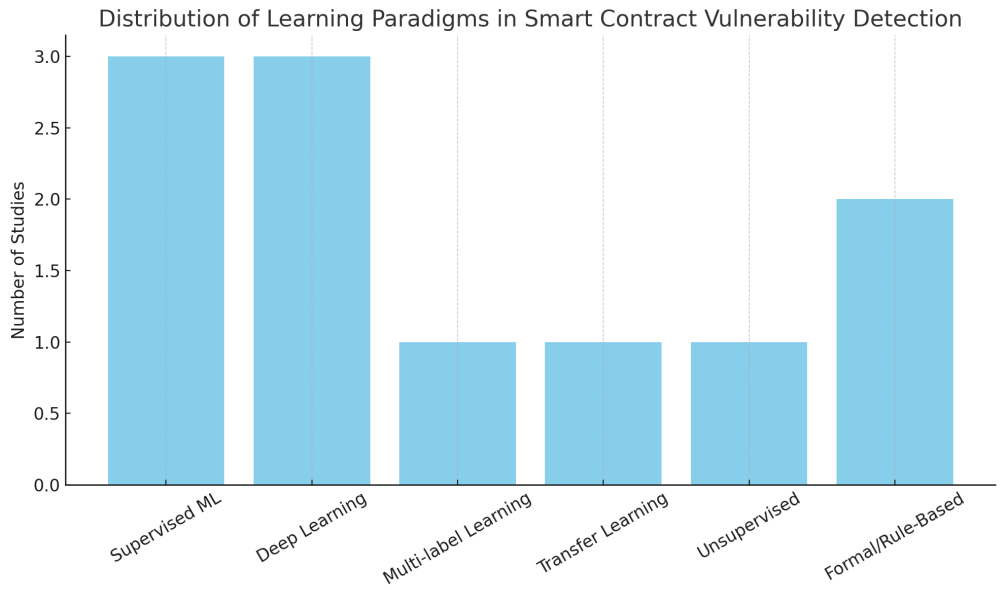

Figure 2.1.1: Learning paradigms in LLMs: supervised, unsupervised, and reinforcement learning approaches that contribute to model capabilities.

2.1.1.1 Evolution of Language Models

The development of LLMs can be traced through several key evolutionary phases. Early statistical approaches, such as n-gram models, provided foundational insights into language structure but were limited by their inability to capture long-range dependencies and semantic relationships. The introduction of neural language models marked a significant advancement, with recurrent neural networks (RNNs) and Long Short-Term Memory (LSTM) networks enabling better sequence modeling capabilities.

The transformer architecture introduced by Vaswani et al. represented a paradigm shift, replacing sequential processing with parallel attention mechanisms. This innovation not only improved training efficiency but also enhanced the model's ability to capture complex linguistic relationships across different scales. Subsequent developments, including the GPT (Generative Pre-trained Transformer) series, BERT (Bidirectional Encoder Representations from Transformers), and other transformer variants, have continued to push the boundaries of what is achievable in natural language understanding and generation.

2.1.1.2 Training Methodologies and Data Requirements

Modern LLMs are trained using vast datasets compiled from diverse sources including web pages, books, academic papers, and other textual resources. The training process typically involves two main phases: pre-training and fine-tuning. During pre-training, models learn general language patterns and world knowledge through self-supervised learning objectives, such as next-token prediction. Fine-tuning, when employed, adapts these general-purpose models to specific tasks or domains using supervised learning on smaller, task-specific datasets.

The scale of training data and computational resources required for state-of-the-art LLMs presents significant challenges for organizations seeking to develop custom models. This has led to the emergence of model-as-a-service offerings and the development of efficient adaptation techniques, such as parameter-efficient fine-tuning and in-context learning, which enable organizations to leverage pre-trained models for their specific use cases without requiring extensive computational resources.

2.1.2 Transformers: The Foundation of Modern LLMs

The Transformer architecture, introduced by Vaswani et al. in their seminal paper "Attention Is All You Need," revolutionized the field of natural language processing and established the foundation for all modern large language models. This architecture addressed fundamental limitations of previous sequence-to-sequence models, particularly the sequential processing bottleneck of recurrent neural networks, by introducing a novel attention mechanism that enables parallel processing of sequence elements.

The core innovation of the Transformer lies in its self-attention mechanism, which computes relationships between all positions in a sequence simultaneously. This approach allows the model to capture long-range dependencies more effectively than previous architectures while enabling efficient parallelization during training. The attention mechanism operates by computing three vectors for each input position: queries (Q), keys (K), and values (V), which are then used to determine the relevance of each position to every other position in the sequence.

2.1.2.1 Attention Mechanisms

The attention function can be mathematically described as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1.1}$$

where d_k is the dimension of the key vectors. This scaled dot-product attention mechanism computes attention weights by taking the dot product of queries and keys, scaling by the square root of the key dimension to prevent extremely small gradients, and applying a softmax function to obtain a probability distribution over the values.

Multi-head attention extends this concept by applying multiple attention functions in parallel, each with different learned linear projections of the input. This allows the model to attend to information from different representation subspaces simultaneously:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \tag{2.1.2}$$

where each head is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{2.1.3}$$

The multi-head mechanism enables the model to capture different types of relationships simultaneously, such as syntactic dependencies, semantic similarities, and discourse-level connections. This parallel processing of multiple attention patterns contributes significantly to the model's ability to understand complex linguistic structures and relationships.

2.1.2.2 Architectural Components

The Transformer architecture consists of an encoder-decoder structure, though many modern LLMs use only the decoder component (as in GPT models) or only the encoder component (as in BERT). Each component is built from a stack of identical layers, with the original paper using 6 layers for both encoder and decoder.

Encoder Layers: Each encoder layer contains two main sub-layers:

- A multi-head self-attention mechanism that allows positions to attend to all positions in the input sequence
- A position-wise fully connected feed-forward network that processes each position independently

Decoder Layers: Each decoder layer contains three sub-layers:

- A masked multi-head self-attention mechanism that prevents positions from attending to subsequent positions
- A multi-head attention mechanism that attends to the encoder output
- A position-wise fully connected feed-forward network

Each sub-layer employs residual connections followed by layer normalization, formulated as:

$$\text{LayerNorm}(x + \text{Sublayer}(x)) \tag{2.1.4}$$

This design choice helps with gradient flow during training and enables the training of deeper networks.

2.1.2.3 Positional Encoding

Since the attention mechanism operates on sets rather than sequences, the Transformer requires explicit positional information to understand the order of tokens. The original paper introduced sinusoidal positional encodings:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{2.1.5}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{2.1.6}$$

where pos is the position, i is the dimension, and d_{model} is the model dimension. These encodings allow the model to learn relative positions and enable generalization to sequences longer than those seen during training.

2.1.2.4 Computational Efficiency and Parallelization

One of the key advantages of the Transformer architecture is its computational efficiency compared to recurrent models. While RNNs require $O(n)$ sequential operations to process a sequence of length n, self-attention layers connect all positions with a constant number of sequentially executed operations. This parallelization capability has been crucial for training large-scale models efficiently.

The computational complexity of self-attention is $O(n^2 \cdot d)$, where n is the sequence length and d is the dimension. While this quadratic complexity can become problematic for very long sequences, various optimization techniques have been developed to address this limitation, including sparse attention patterns, linear attention mechanisms, and hierarchical attention structures.

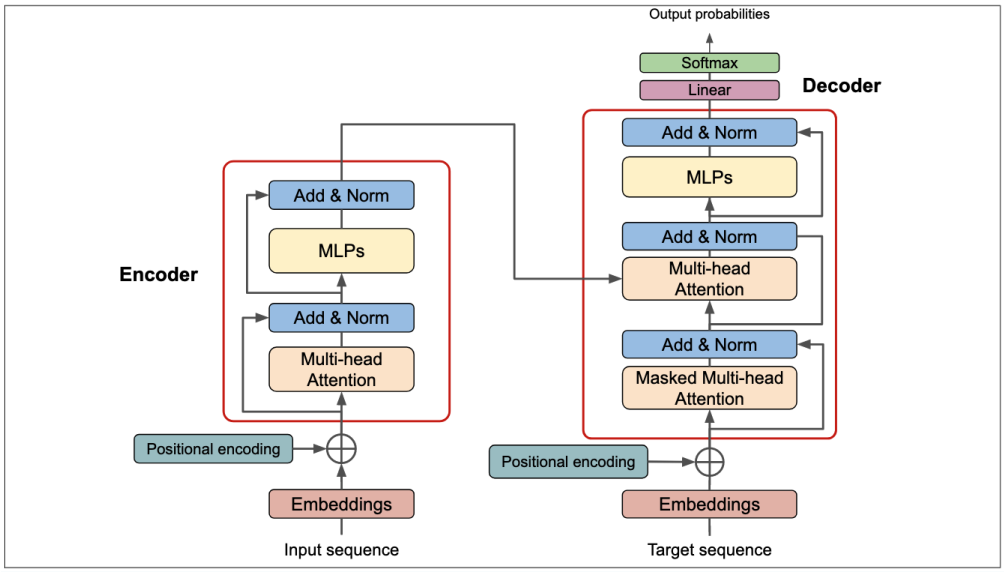

Figure 2.1.2: The Transformer architecture highlighting the encoder-decoder structure with multi-head self-attention and feed-forward layers. The parallel processing capability of attention mechanisms enables efficient training and inference.

2.1.2.5 Impact on Modern LLM Development

The Transformer architecture has become the de facto standard for large language models, with virtually all state-of-the-art models building upon its foundations. The scalability of the attention mechanism has enabled the development of increasingly large models, from the original Transformer with millions of parameters to contemporary models with hundreds of billions of parameters.

The architecture's flexibility has also led to numerous variants optimized for different use cases, including encoder-only models like BERT for understanding tasks, decoder-only models like GPT for generation tasks, and encoder-decoder models like T5 for sequence-to-sequence tasks. Each variant leverages the core attention mechanism while adapting the overall architecture to specific requirements.

2.1.3 GPT-3: Scaling Laws and Few-Shot Learning Capabilities

GPT-3 (Generative Pre-trained Transformer 3), developed by OpenAI, represents a landmark achievement in the scaling of large language models and has fundamentally changed our understanding of what is possible with language AI. With 175 billion parameters, GPT-3 demonstrated that simply scaling up model size, training data, and computational resources could lead to qualitatively different capabilities, including few-shot learning, task generalization, and emergent abilities that were not explicitly programmed.

The development of GPT-3 was guided by empirical observations of scaling laws in language models, which suggest that model performance improves predictably with increases in model size, dataset size, and computational budget. These scaling laws provided the theoretical foundation for investing in increasingly large models, leading to the breakthrough capabilities observed in GPT-3.

2.1.3.1 Architecture and Scale

GPT-3 employs a decoder-only transformer architecture, similar to its predecessors GPT and GPT-2, but at an unprecedented scale. The model consists of 96 transformer layers, each with 96 attention

heads and a hidden dimension of 12,288. This massive architecture enables the model to capture incredibly complex patterns in language and maintain coherent context over long sequences.

The training process utilized a diverse dataset comprising Common Crawl, WebText2, Books1, Books2, and Wikipedia, totaling approximately 570GB of text data after filtering and preprocessing. This dataset diversity is crucial for the model's broad capabilities, as it ensures exposure to various writing styles, domains, and types of knowledge.

Key architectural specifications of GPT-3 include:

- **Parameters:** 175 billion parameters distributed across layers
- **Context Window:** 2,048 tokens, allowing for substantial context retention
- **Vocabulary Size:** 50,257 tokens using byte-pair encoding
- **Training Compute:** Approximately 3.14×10^{23} FLOPs
- **Model Parallelism:** Distributed training across multiple GPUs using both data and model parallelism

2.1.3.2 Few-Shot Learning Paradigm

One of GPT-3's most remarkable capabilities is its ability to perform few-shot learning, where the model can adapt to new tasks with just a few examples provided in the input prompt. This paradigm differs fundamentally from traditional machine learning approaches that require extensive task-specific training data and fine-tuning procedures.

GPT-3's few-shot learning operates through three main modalities:

Zero-shot Learning: The model performs tasks based solely on natural language descriptions without any examples. For instance, when given the instruction "Translate the following English text to French," GPT-3 can perform translation without seeing any translation examples.

One-shot Learning: The model receives a single example of the desired task format before being asked to perform the task on new input. This single example helps establish the expected input-output pattern.

Few-shot Learning: The model is provided with a small number of examples (typically 2-64) that demonstrate the task format and expected behavior. This approach often yields the best performance across various tasks.

The few-shot learning capability emerges from the model's extensive pre-training on diverse text data, which enables it to recognize patterns and adapt to new contexts without explicit parameter updates. This in-context learning ability has significant implications for enterprise applications, as it reduces the need for task-specific model training and enables rapid deployment of AI solutions for new use cases.

2.1.3.3 Performance Across Diverse Tasks

GPT-3's evaluation across numerous NLP benchmarks demonstrated its versatility and effectiveness across a wide range of tasks. The model showed strong performance in:

Language Modeling and Text Completion: GPT-3 achieved state-of-the-art results on several language modeling benchmarks, including a 76

Question Answering: On TriviaQA, GPT-3 achieved 64.3

Translation Tasks: While zero-shot translation performance was modest, few-shot learning with just a single example improved performance by over 7 BLEU points, approaching competitive results with specialized translation models.

Common Sense Reasoning: GPT-3 demonstrated strong performance on tasks requiring common sense reasoning, such as the Winograd Schema Challenge (88.3

Arithmetic and Symbolic Reasoning: The model showed emergent mathematical abilities, performing multi-digit arithmetic and solving algebraic problems, though performance degraded with increasing complexity.

2.1.3.4 Scaling Laws and Emergent Abilities

The development of GPT-3 was informed by empirical scaling laws that describe the relationship between model performance and three key factors: model size (number of parameters), dataset size, and computational budget. These laws suggest that performance improvements follow predictable power-law relationships, enabling informed decisions about resource allocation for model development.

The scaling laws reveal several important insights:

- Model performance improves smoothly and predictably with scale across multiple orders of magnitude
- Larger models are more sample-efficient, achieving better performance with less training data
- Computational optimal training involves scaling model size and training data proportionally
- Performance gains from scaling appear to continue without obvious saturation points

GPT-3 also exhibited several emergent abilities that were not present in smaller models, including:

- Sophisticated few-shot learning across diverse domains
- Complex reasoning and problem-solving capabilities
- Creative writing and storytelling abilities
- Code generation and programming assistance
- Multi-step logical reasoning and mathematical problem solving

These emergent abilities suggest that scale alone can lead to qualitatively new capabilities, providing a strong incentive for continued scaling of language models.

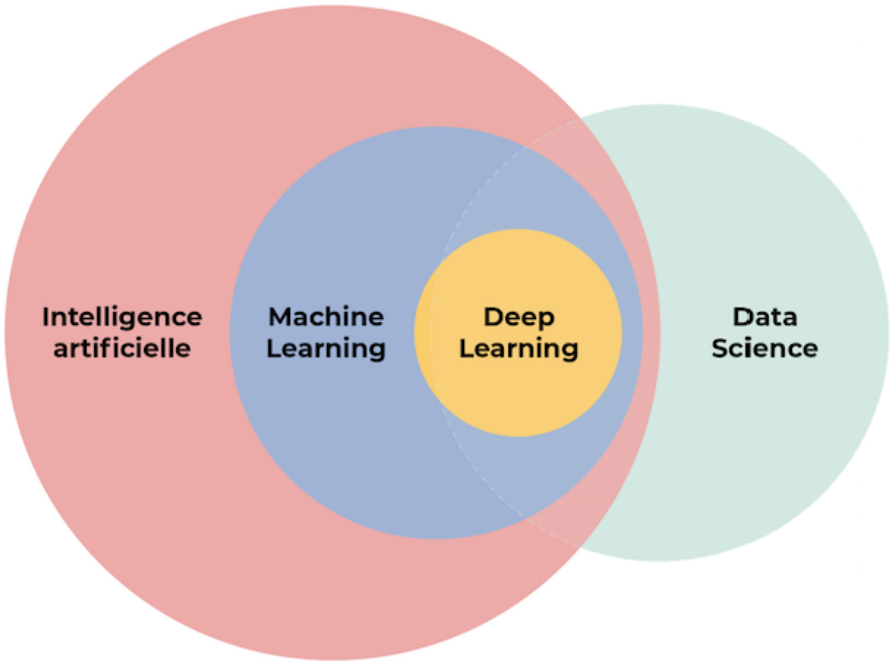

Figure 2.1.3: Applications of GPT-3 across various AI fields, demonstrating the model's versatility in handling diverse tasks without task-specific training.

2.1.3.5 Implications for Enterprise Integration

GPT-3's capabilities have significant implications for enterprise applications and integration patterns. The model's few-shot learning abilities enable rapid deployment of AI solutions without extensive training procedures, making it particularly suitable for dynamic business environments where requirements change frequently.

Key enterprise implications include:

- **Reduced Time-to-Deployment:** Few-shot learning eliminates the need for extensive training data collection and model fine-tuning
- **Versatile Applications:** A single model can handle multiple business functions, from customer service to content generation
- **Cost-Effective AI Solutions:** Organizations can leverage powerful AI capabilities without investing in custom model development
- **Scalable Integration:** The model's API-based access enables easy integration into existing enterprise systems

However, the scale and computational requirements of GPT-3 also present challenges for enterprise deployment, including high inference costs, latency concerns, and the need for robust integration architectures that can handle the model's resource requirements while maintaining system performance and reliability.

2.1.4 Knowledge Distillation and Model Optimization

While large language models like GPT-3 demonstrate remarkable capabilities, their computational requirements and resource consumption pose significant challenges for practical deployment, particularly in resource-constrained environments such as edge devices, mobile applications, and real-time

systems. Knowledge distillation has emerged as a crucial technique for addressing these challenges by creating smaller, more efficient models that retain much of the performance of their larger counterparts.

Knowledge distillation, first introduced by Hinton et al., involves training a smaller "student" model to mimic the behavior of a larger "teacher" model. This process transfers the knowledge embedded in the teacher model to the student model, often resulting in compact models that achieve comparable performance while requiring significantly fewer computational resources.

2.1.4.1 DiXtill: XAI-Driven Knowledge Distillation

DiXtill represents an innovative approach to knowledge distillation that incorporates explainable artificial intelligence (XAI) techniques to enhance both the efficiency and interpretability of distilled models. This method addresses a critical limitation of traditional distillation approaches by ensuring that the student model not only mimics the teacher's predictions but also learns to generate similar explanations for its decisions.

The DiXtill methodology consists of several key components:

Explainer Integration: The framework incorporates local explanation techniques to guide the distillation process. These explanations help the student model understand not just what predictions to make, but why those predictions are appropriate, leading to more robust and interpretable behavior.

Self-Explainable Architecture: The student model is designed as a bi-directional LSTM network enhanced with masked attention mechanisms. This architecture enables the model to provide explanations for its predictions while maintaining computational efficiency.

Attention-Based Learning: The attention mechanism allows the student model to focus on the most relevant parts of the input, similar to how the teacher model processes information. This attention-based approach improves both performance and interpretability.

Multi-Objective Training: The training process optimizes for both prediction accuracy and explanation consistency, ensuring that the distilled model maintains the teacher's reasoning patterns while achieving efficient performance.

2.1.4.2 Distillation Process and Methodology

The DiXtill distillation process follows a structured approach that combines traditional knowledge distillation with explainability constraints:

Teacher Model Preparation: The process begins with a pre-trained teacher model (such as FinBERT) that has been fine-tuned for the target task. The teacher model serves as the source of both predictions and explanations.

Explanation Generation: Local explanation techniques are applied to the teacher model to generate word-level attributions for different classes. These explanations reveal which parts of the input are most important for the model's decisions.

Student Architecture Design: The student model is implemented as a bi-directional LSTM with masked attention layers. This architecture is significantly more compact than transformer-based models while still capable of capturing sequential dependencies and generating attention-based explanations.

Joint Training Objective: The student model is trained using a combined loss function that includes:

- Traditional distillation loss to match teacher predictions
- Explanation consistency loss to align attention patterns with teacher explanations
- Task-specific loss for the target application

Evaluation and Refinement: The distilled model is evaluated on both performance metrics and explanation quality, with iterative refinement to optimize the balance between efficiency and accuracy.

2.1.4.3 Performance and Efficiency Gains

Experimental results from DiXtill demonstrate significant improvements in both computational efficiency and model interpretability compared to traditional distillation approaches:

Compression Ratios: DiXtill achieves superior compression ratios compared to other model compression techniques such as post-training quantization (PTQ) and attention head pruning (AHP). The method can reduce model size by orders of magnitude while maintaining competitive performance.

Inference Speedup: The lightweight architecture of the student model enables significant speedup in inference time, making it suitable for real-time applications and resource-constrained environments.

Explanation Quality: The XAI-driven approach ensures that the student model produces explanations that are consistent with the teacher model, as measured by explanation agreement metrics. This consistency is crucial for applications requiring interpretable AI decisions.

Performance Retention: Despite the significant reduction in model size, DiXtill maintains performance levels close to the teacher model across various metrics including accuracy, macro F1 score, Matthews correlation coefficient, and AUC.

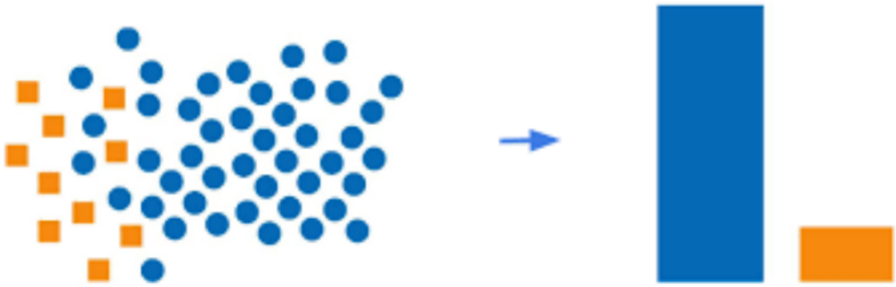

Figure 2.1.4: Model distillation process showing the knowledge transfer from a large teacher model to a compact student model while maintaining performance and interpretability.

2.1.4.4 Enterprise Applications and Deployment Scenarios

The combination of efficiency and interpretability offered by DiXtill makes it particularly suitable for enterprise applications where both performance and explainability are critical requirements:

Financial Services: In financial sentiment analysis and risk assessment, DiXtill enables deployment of AI models that can provide both accurate predictions and clear explanations for regulatory compliance and decision transparency.

Healthcare Applications: Medical diagnosis and treatment recommendation systems benefit from the interpretable nature of distilled models, allowing healthcare professionals to understand and validate AI-generated insights.

Edge Computing: The compact size and efficient inference of DiXtill models make them ideal for deployment on edge devices, enabling AI capabilities in IoT applications, mobile devices, and offline environments.

Real-time Systems: Applications requiring low-latency responses, such as chatbots, recommendation systems, and automated customer service, can leverage the speed advantages of distilled models while maintaining quality.

2.1.4.5 Integration with Enterprise Protocols

The characteristics of distilled models like those produced by DiXtill have specific implications for integration with enterprise protocols:

REST API Integration: The lightweight nature of distilled models makes them well-suited for REST API deployment, where stateless request-response patterns can efficiently handle model inference without significant computational overhead.

Streaming Protocol Compatibility: The fast inference speed of distilled models enables real-time processing in streaming applications, where continuous data processing and immediate response generation are required.

MCP Integration: The interpretable nature of DiXtill models aligns well with Model Context Protocol requirements, where explanation metadata can be included in model responses to provide transparency and context for AI decisions.

2.1.5 Enterprise Integration Protocols for LLM Systems

The integration of large language models into enterprise systems requires careful consideration of communication protocols that can effectively handle the unique characteristics and requirements of AI-powered applications. The choice of integration protocol significantly impacts system performance, scalability, maintainability, and user experience. This section examines three primary protocol approaches for LLM integration: REST (Representational State Transfer), MCP (Model Context Protocol), and Streaming Protocols.

Each protocol offers distinct advantages and trade-offs in terms of latency, throughput, resource utilization, and architectural complexity. Understanding these characteristics is essential for making informed decisions about LLM integration strategies that align with specific enterprise requirements and constraints.

2.1.5.1 REST (Representational State Transfer) Protocol

REST has emerged as the dominant architectural style for web services and API design, offering a standardized approach to system integration based on stateless client-server communication. In the context of LLM integration, REST provides a familiar and widely supported framework for accessing AI capabilities through well-defined API endpoints.

Core Principles of REST for LLM Integration:

Stateless Communication: Each request to an LLM service contains all necessary information for processing, eliminating server-side state management. This principle simplifies deployment and scaling but may require including extensive context in each request.

Uniform Interface: REST APIs provide standardized methods (GET, POST, PUT, DELETE) for interacting with LLM services, enabling consistent integration patterns across different applications and use cases.

Resource-Based Architecture: LLM capabilities are exposed as resources (e.g., /completion, /translation, /summarization) that can be accessed through HTTP methods, providing intuitive and discoverable interfaces.

Cacheable Responses: REST's caching mechanisms can be leveraged to store and reuse LLM responses for identical inputs, reducing computational costs and improving response times for repeated queries.

Advantages of REST for LLM Integration:

- **Simplicity and Familiarity:** REST's widespread adoption means that developers have extensive experience with the protocol, reducing integration complexity and time-to-market.

- **Scalability:** The stateless nature of REST enables horizontal scaling of LLM services across multiple instances without complex coordination mechanisms.
- **Tooling and Infrastructure:** Extensive ecosystem of tools for API development, testing, monitoring, and management supports REST-based LLM integrations.
- **Security and Authentication:** Well-established security patterns for REST APIs, including OAuth, JWT, and API key authentication, provide robust protection for LLM services.
- **Interoperability:** REST's platform-agnostic nature enables integration across diverse technology stacks and programming languages.

Challenges and Limitations:

- **Latency Overhead:** HTTP request-response cycles introduce latency, particularly problematic for real-time applications requiring immediate LLM responses.
- **Context Management:** Stateless communication requires including complete context in each request, potentially leading to large payloads and inefficient data transfer.
- **Limited Real-time Capabilities:** Traditional REST is not well-suited for scenarios requiring continuous interaction or streaming responses from LLMs.
- **Resource Inefficiency:** Each request establishes a new connection, leading to overhead in scenarios with frequent, small interactions with LLM services.

2.1.5.2 Model Context Protocol (MCP)

The Model Context Protocol represents a novel approach to LLM integration specifically designed to address the unique requirements of AI-powered applications. MCP focuses on efficient context management, dynamic model interaction, and optimized communication patterns that align with the characteristics of large language models.

Key Features of MCP:

Context-Aware Communication: MCP maintains session state and context across multiple interactions, enabling more efficient communication for multi-turn conversations and complex reasoning tasks.

Dynamic Model Selection: The protocol supports intelligent routing to different models based on task requirements, enabling optimization of resource utilization and response quality.

Incremental Context Updates: Rather than sending complete context with each request, MCP supports incremental updates that reduce bandwidth usage and improve response times.

Metadata-Rich Responses: MCP responses include extensive metadata about model confidence, reasoning steps, and alternative interpretations, supporting explainable AI requirements.

Adaptive Batching: The protocol can intelligently batch multiple requests to optimize throughput while maintaining acceptable latency for individual requests.

Advantages of MCP for LLM Integration:

- **Optimized for AI Workloads:** MCP is specifically designed for the communication patterns and requirements of LLM applications, offering better performance characteristics than general-purpose protocols.
- **Efficient Context Handling:** By maintaining state and supporting incremental updates, MCP reduces the overhead associated with context management in conversational AI applications.
- **Enhanced Explainability:** The protocol's support for metadata-rich responses enables better integration with enterprise requirements for interpretable AI systems.

- **Resource Optimization:** Dynamic model selection and adaptive batching help optimize computational resource utilization across different types of AI tasks.
- **Flexible Interaction Patterns:** MCP supports both synchronous and asynchronous communication patterns, enabling diverse integration scenarios.

Challenges and Considerations:

- **Novelty and Adoption:** As a newer protocol, MCP has limited ecosystem support and requires specialized knowledge for implementation and maintenance.
- **Complexity:** The advanced features of MCP introduce additional complexity in system design and debugging compared to simpler protocols.
- **Standardization:** The protocol is still evolving, with potential changes that could impact long-term compatibility and stability.
- **Vendor Lock-in:** Limited implementations may create dependency on specific vendors or technologies.

2.1.5.3 Streaming Protocols

Streaming protocols enable real-time, continuous data exchange between clients and LLM services, making them particularly suitable for applications requiring immediate responses and interactive experiences. These protocols support scenarios where LLM responses are generated incrementally, allowing users to see partial results as they become available.

Types of Streaming Protocols for LLM Integration:

WebSocket-based Streaming: Provides full-duplex communication channels that enable real-time interaction with LLM services, supporting both text input and incremental response streaming.

Server-Sent Events (SSE): Offers a simpler alternative for scenarios where only server-to-client streaming is required, such as displaying incremental text generation from LLMs.

HTTP/2 and HTTP/3 Streaming: Leverages modern HTTP protocols' streaming capabilities to enable efficient, multiplexed communication with LLM services.

gRPC Streaming: Provides high-performance streaming communication with strong typing and efficient serialization, suitable for high-throughput LLM applications.

Advantages of Streaming Protocols:

- **Real-time Responsiveness:** Streaming enables immediate display of partial results, improving user experience in interactive applications.
- **Reduced Perceived Latency:** Users can begin reading or processing LLM responses before the complete output is generated, reducing perceived wait times.
- **Efficient Resource Utilization:** Streaming can reduce memory usage by processing and transmitting data incrementally rather than buffering complete responses.
- **Interactive Experiences:** Supports conversational AI applications where continuous interaction and immediate feedback are essential.
- **Scalable Communication:** Modern streaming protocols can handle many concurrent connections efficiently, supporting large-scale applications.

Challenges and Limitations:

- **Complexity:** Streaming protocols require more sophisticated client and server implementations, including connection management and error handling.

- **Infrastructure Requirements:** Supporting streaming may require additional infrastructure components such as load balancers and proxies that can handle persistent connections.
- **Error Handling:** Managing errors and connection failures in streaming scenarios is more complex than in request-response patterns.
- **Debugging and Monitoring:** Troubleshooting streaming applications requires specialized tools and techniques for analyzing real-time data flows.

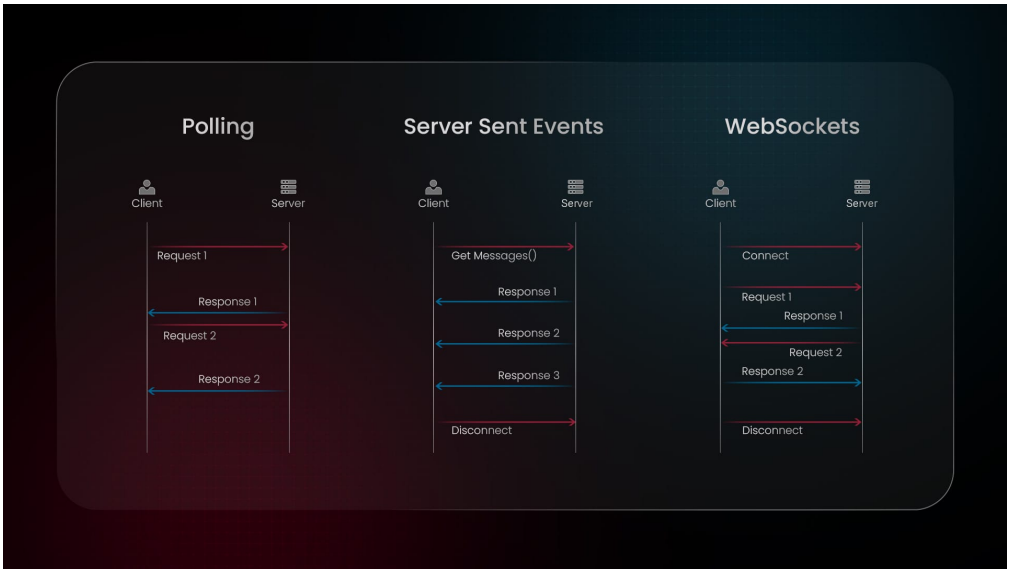

Figure 2.1.5: Execution flow in streaming protocols for real-time LLM applications, showing the incremental response generation and client-server communication patterns.

2.1.6 Comparative Analysis of Integration Protocols

The selection of an appropriate integration protocol for LLM deployment significantly impacts system performance, user experience, and operational efficiency. Each protocol offers distinct advantages and trade-offs that must be carefully evaluated against specific enterprise requirements and constraints.

Property	REST	MCP	Streaming
Latency	Moderate (HTTP overhead)	Low (context-aware)	Very Low (real-time)
Throughput	High (cacheable)	High (batched)	Moderate (persistent)
Scalability	Excellent (stateless)	Good (session mgmt)	Good (connection pool)
State Management	Stateless	Context-Aware	Stateful
Resource Usage	Moderate	Optimized	Variable
Implementation Complexity	Low	Medium	High
Ecosystem Support	Excellent	Limited	Good
Real-time Capability	Poor	Moderate	Excellent
Error Handling	Simple	Advanced	Complex
Security	Mature	Evolving	Established

Table 1: Comprehensive comparison of integration protocols for LLMs in enterprise applications, evaluating key performance and operational characteristics.

2.1.6.1 Performance Characteristics Analysis

Latency Considerations: REST protocols introduce inherent latency due to HTTP request-response cycles, connection establishment overhead, and the need to include complete context in each request. For applications requiring sub-second response times, this overhead can significantly impact user experience. MCP addresses some of these limitations through persistent connections and incremental context updates, while streaming protocols offer the lowest latency for interactive applications.

Throughput and Scalability: REST's stateless nature enables excellent horizontal scaling and load distribution, making it suitable for high-throughput applications. The caching capabilities of HTTP can further improve throughput for repeated queries. MCP's batching mechanisms can optimize throughput for multiple concurrent requests, while streaming protocols may have lower peak throughput due to persistent connection overhead.

Resource Utilization: Different protocols have varying impacts on computational and network resources. REST's stateless design minimizes server-side resource requirements but may result in redundant data transmission. MCP's context management can reduce network traffic but requires additional memory for session state. Streaming protocols maintain persistent connections, which can be resource-intensive at scale but enable efficient real-time communication.

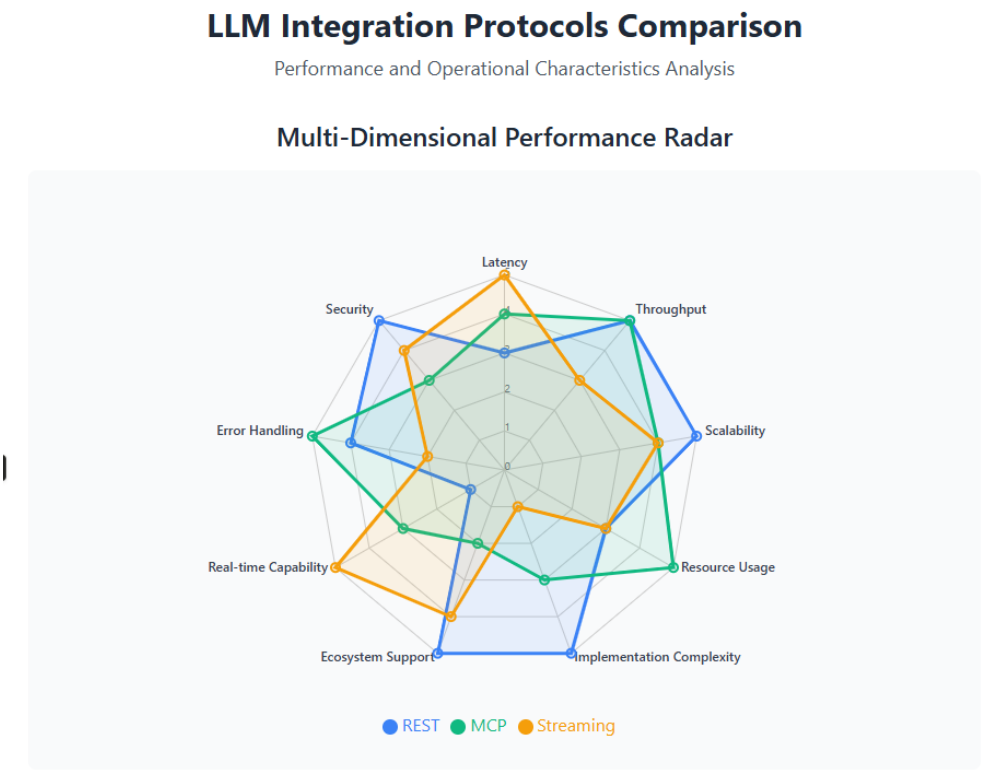

Figure 2.1.6: Performance comparison across different protocols showing latency, throughput, and resource utilization characteristics under various load conditions.

2.1.7 Technical Challenges in LLM Integration

The integration of large language models into enterprise systems presents numerous technical challenges that must be addressed to ensure reliable, secure, and efficient operation. These challenges span multiple domains including computational efficiency, system reliability, security, and interoperability.

2.1.7.1 Computational and Resource Management Challenges

Memory and Storage Requirements: Modern LLMs require substantial memory resources for inference, with models like GPT-3 requiring hundreds of gigabytes of GPU memory for optimal performance. Enterprise deployments must consider:

- **Memory Optimization:** Techniques such as model sharding, quantization, and gradient checkpointing to reduce memory footprint
- **Storage Management:** Efficient model loading and caching strategies to minimize startup times and storage costs
- **Dynamic Scaling:** Auto-scaling mechanisms that can adapt to varying demand while managing resource costs
- **Multi-tenancy:** Strategies for sharing computational resources across multiple applications and users while maintaining isolation

Inference Optimization: The computational cost of LLM inference presents significant challenges for enterprise deployment:

- **Batching Strategies:** Dynamic batching to optimize throughput while maintaining acceptable latency
- **Model Parallelism:** Distributing model computation across multiple GPUs or nodes
- **Caching and Memoization:** Intelligent caching of intermediate results and common responses
- **Hardware Acceleration:** Leveraging specialized hardware such as TPUs, FPGAs, and inference-optimized chips

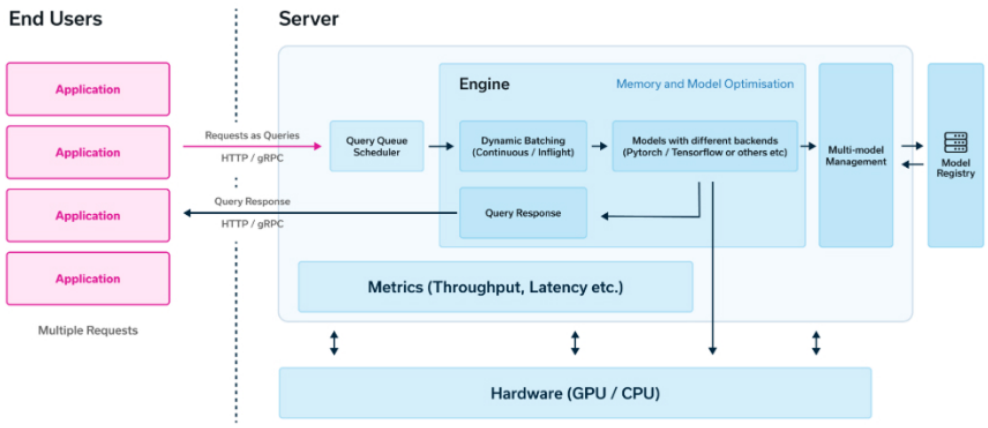

Figure 2.1.7: Enterprise resource management architecture for LLM deployment, showing distributed computing, memory optimization, and auto-scaling components.

2.1.7.2 Latency and Real-time Performance

Enterprise applications often require real-time or near-real-time responses from LLM systems, presenting several performance challenges:

End-to-End Latency Optimization:

- **Network Latency:** Minimizing network overhead through edge deployment and optimized protocols
- **Model Inference Time:** Reducing computational latency through model optimization and efficient serving infrastructure
- **Queue Management:** Implementing intelligent request queuing and prioritization systems
- **Warm-up Strategies:** Maintaining model readiness to avoid cold-start delays

Predictive Performance Management:

- **Load Forecasting:** Predicting demand patterns to pre-scale resources
- **Performance Monitoring:** Real-time monitoring of latency and throughput metrics
- **Adaptive Optimization:** Dynamic adjustment of system parameters based on performance feedback
- **SLA Management:** Ensuring consistent performance levels to meet service level agreements

2.1.7.3 Security and Privacy Considerations

The integration of LLMs into enterprise systems raises significant security and privacy concerns that must be addressed through comprehensive security frameworks:

Data Protection and Privacy:

- **Input Sanitization:** Protecting against prompt injection attacks and malicious inputs
- **Output Filtering:** Preventing the generation of inappropriate or sensitive content
- **Data Encryption:** Ensuring end-to-end encryption of sensitive data in transit and at rest
- **Privacy Preservation:** Implementing techniques such as differential privacy and federated learning

Access Control and Authentication:

- **Multi-factor Authentication:** Strong authentication mechanisms for accessing LLM services
- **Role-based Access Control:** Granular permissions for different user roles and applications
- **API Security:** Secure API design with proper rate limiting and abuse prevention
- **Audit Logging:** Comprehensive logging of all interactions for security monitoring and compliance

Model Security:

- **Model Protection:** Preventing unauthorized access to model parameters and architecture
- **Adversarial Robustness:** Defending against adversarial attacks designed to manipulate model behavior
- **Model Versioning:** Secure model deployment and rollback mechanisms
- **Compliance:** Ensuring adherence to industry regulations and data protection standards

2.1.7.4 Interoperability and System Integration

Enterprise environments typically involve complex, heterogeneous technology stacks that must seamlessly integrate with LLM services:

API Design and Standardization:

- **Protocol Compatibility:** Supporting multiple communication protocols to accommodate diverse client applications
- **Data Format Standardization:** Consistent data exchange formats across different systems and platforms
- **Version Management:** Backward compatibility and graceful migration strategies for API updates
- **Documentation and SDKs:** Comprehensive documentation and software development kits for multiple programming languages

Legacy System Integration:

- **Adapter Patterns:** Bridging LLM capabilities with existing enterprise systems

- **Data Pipeline Integration:** Seamless integration with existing data processing and analytics pipelines
- **Workflow Orchestration:** Incorporating LLM services into existing business process workflows
- **Monitoring Integration:** Integrating LLM monitoring with existing observability platforms

2.1.8 Scalability and Performance Optimization

Scaling LLM services to meet enterprise demands requires sophisticated approaches to performance optimization, resource management, and system architecture. This section examines key strategies and techniques for achieving scalable LLM deployments.

2.1.8.1 Horizontal and Vertical Scaling Strategies

Horizontal Scaling Approaches: Horizontal scaling involves distributing LLM workloads across multiple compute instances or nodes:

- **Model Replication:** Deploying multiple instances of the same model across different servers to handle increased request volume
- **Load Balancing:** Intelligent distribution of requests across model instances using various algorithms (round-robin, least connections, weighted distribution)
- **Geographic Distribution:** Deploying models across multiple regions to reduce latency and improve availability
- **Auto-scaling:** Dynamic scaling based on metrics such as CPU utilization, memory usage, and request queue length

Vertical Scaling Considerations: Vertical scaling involves increasing the computational capacity of individual nodes:

- **GPU Scaling:** Adding more powerful GPUs or increasing GPU memory to handle larger models
- **Memory Optimization:** Increasing system memory to support larger batch sizes and reduce I/O overhead
- **Storage Performance:** Utilizing high-performance storage systems for faster model loading and data access
- **Network Bandwidth:** Ensuring adequate network capacity for high-throughput applications

2.1.8.2 Performance Monitoring and Optimization

Effective performance monitoring is crucial for maintaining optimal LLM system performance and identifying optimization opportunities:

Key Performance Metrics:

- **Latency Metrics:** P50, P95, and P99 response times for different types of requests
- **Throughput Metrics:** Requests per second, tokens per second, and concurrent user capacity

- **Resource Utilization:** CPU, GPU, memory, and network utilization across all system components
- **Error Rates:** HTTP error rates, timeout rates, and model inference failures
- **Queue Metrics:** Request queue depth, wait times, and processing times

Optimization Techniques:

- **Dynamic Batching:** Optimizing batch sizes based on current load and latency requirements
- **Model Warm-up:** Pre-loading models and maintaining ready instances to reduce cold-start latency
- **Caching Strategies:** Implementing multi-level caching for common queries and intermediate results
- **Request Prioritization:** Implementing priority queues for different types of requests and users

2.1.9 Emerging Trends and Future Directions

The field of LLM integration is rapidly evolving, with several emerging trends and technologies that will shape the future of enterprise AI deployments. Understanding these trends is crucial for making strategic decisions about LLM integration architectures.

2.1.9.1 Edge Computing and Distributed Inference

The deployment of LLMs at the edge represents a significant trend toward reducing latency and improving data privacy:

Edge Deployment Strategies:

- **Model Compression:** Advanced techniques for reducing model size while maintaining performance
- **Federated Learning:** Training models across distributed edge devices while preserving data privacy
- **Hybrid Architectures:** Combining edge inference for simple tasks with cloud processing for complex queries
- **Adaptive Model Selection:** Dynamically choosing between edge and cloud processing based on requirements

Technical Challenges and Solutions:

- **Resource Constraints:** Developing efficient models that can run on resource-limited edge devices
- **Connectivity Issues:** Designing systems that can operate effectively with intermittent connectivity
- **Model Updates:** Implementing efficient mechanisms for updating edge-deployed models
- **Quality Assurance:** Ensuring consistent performance across diverse edge deployment environments

2.1.9.2 Multimodal Integration

The evolution toward multimodal LLMs that can process text, images, audio, and other data types presents new integration challenges and opportunities:

Multimodal Protocol Requirements:

- **Data Format Support:** Protocols must handle diverse data types efficiently
- **Bandwidth Optimization:** Managing the increased bandwidth requirements of multimodal data
- **Processing Coordination:** Synchronizing processing of different modalities
- **Result Integration:** Combining outputs from different modalities into coherent responses

Enterprise Applications:

- **Document Analysis:** Processing documents containing text, images, and tables
- **Customer Service:** Handling voice, text, and visual inputs in support systems
- **Content Creation:** Generating multimedia content for marketing and communication
- **Data Analytics:** Analyzing complex datasets containing multiple data types

2.1.9.3 Specialized Hardware and Acceleration

The development of specialized hardware for AI workloads is creating new opportunities for LLM optimization:

Hardware Acceleration Technologies:

- **AI Chips:** Purpose-built processors optimized for transformer architectures
- **Neuromorphic Computing:** Hardware that mimics brain-like processing for efficiency gains
- **Quantum Computing:** Potential future applications of quantum processors for certain AI tasks
- **Optical Computing:** Using light-based processing for high-speed, low-power inference

Integration Implications:

- **Protocol Adaptation:** Modifying protocols to leverage hardware-specific optimizations
- **Deployment Strategies:** Adapting deployment architectures for heterogeneous hardware environments
- **Performance Optimization:** Developing new optimization techniques for specialized hardware
- **Cost Management:** Balancing performance gains with hardware acquisition and operational costs

2.1.10 Conclusion

This comprehensive background chapter has examined the fundamental technologies, architectures, and integration challenges associated with large language models in enterprise environments. The analysis of Transformers, GPT-3, and knowledge distillation techniques provides the foundation for understanding how these powerful AI systems can be effectively integrated into business applications.

The examination of integration protocols—REST, MCP, and Streaming—reveals the complex trade-offs between performance, scalability, and implementation complexity that organizations must consider when deploying LLM solutions. Each protocol offers distinct advantages for different use cases, and the choice of integration approach significantly impacts system performance and user experience.

The technical challenges discussed, including computational efficiency, security, scalability, and interoperability, highlight the multifaceted nature of enterprise LLM deployment. Addressing these challenges requires sophisticated approaches to system architecture, resource management, and performance optimization.

Looking toward the future, emerging trends such as edge computing, multimodal integration, and specialized hardware acceleration will continue to reshape the landscape of LLM integration. Organizations that understand these trends and prepare for their implications will be better positioned to leverage the full potential of large language models in their enterprise applications.

The subsequent chapters will build upon this foundation to present practical implementations and empirical evaluations of different integration approaches, providing concrete guidance for organizations seeking to optimize their LLM integration strategies. The comparative study of REST, MCP, and Streaming protocols will demonstrate how theoretical considerations translate into real-world performance characteristics and operational requirements.

Part 3

State of The Art

3.1 Introduction

Smart contracts have become a critical component of decentralized applications, enabling autonomous execution of logic on blockchain platforms. However, their immutability and complexity make them prone to security vulnerabilities, with incidents such as the DAO hack highlighting the risks of deploying unaudited or flawed contracts.

To address this, various vulnerability detection tools have been developed—ranging from rule-based static analyzers like Oyente and Securify to symbolic execution engines like Mythril. While these tools provide valuable insights, they often face limitations in scalability, precision, and adaptability to new threats.

As a result, researchers have begun exploring artificial intelligence techniques to enhance vulnerability detection. Machine learning models can learn from labeled contract datasets to identify risky patterns, while deep learning approaches—including CNNs, RNNs, and Graph Neural Networks—enable more advanced analysis of smart contract structures, such as opcodes and control flow graphs.

Recent work also incorporates transformer-based models for source code understanding and vulnerability prediction. Despite these advancements, challenges like data quality, class imbalance, and explainability remain open issues.

This chapter reviews key contributions and methodologies in this domain, laying the groundwork for the proposed AI-based approach presented in this thesis.

3.2 Related Work

3.2.1 Used Datasets

The progress in smart contract vulnerability detection has been largely fueled by the availability of public datasets that contain real-world or labeled smart contracts. These datasets enable the development, training, and benchmarking of AI-based models for static or semantic vulnerability analysis. Below is an overview of the most widely used datasets in the field.

Public Datasets

- **SmartBugs Dataset** [26]: This benchmark dataset contains over 47,000 verified smart contracts from Etherscan. Contracts are labeled using several static analyzers like Oyente, Securify, and Mythril. It is widely used for training and evaluating AI-based vulnerability detection systems.
- **Dataset Used in Feng et al.** [28]: Feng et al. collected and processed 43,937 verified contracts from Etherscan and labeled them using Oyente to detect six vulnerability types. These contracts were later transformed into opcode sequences and tokenized into n-grams for input into machine learning models.
- **Lesimple's Contract Corpus** [34]: In this work, smart contracts were scraped from Etherscan and annotated manually or using a mix of tools. This dataset was used to train and evaluate deep learning models such as LSTMs and CNNs on opcode sequences.
- **Zhang et al. (SCVD Dataset)** [29]: This dataset includes over 20,000 smart contracts collected from Etherscan and labeled using multiple tools, including Slither, SmartCheck, and manual inspection. It supports multi-label classification tasks with annotations for several vulnerabilities per contract.
- **Albert et al. (SAFEVM Set)** [35]: More than 24,000 smart contracts were collected for verification using a formal translation into C and analysis via tools such as CPAchecker and SeaHorn. The dataset focuses on bytecode-level execution safety.
- **Tann et al. (EtherCorpus)** [36]: EtherCorpus includes over 160,000 smart contracts with both source code and compiled bytecode. The dataset supports embedding learning and is often used with transformer-based models or opcode token analysis.

See table 2 for a comparaison between the mentioned datasets.

Study	Year	Source	Samples	Balanced
SmartBugs [26]	2020	Contracts from Etherscan, labeled with Mythril, Oyente, Securify	47,000+	No
Feng et al. [28]	2024	Verified contracts from Etherscan, labeled via Oyente	43,937	No
Lesimple [34]	2020	Etherscan contracts annotated manually and with tools	5,000	No
Zhang et al. (SCVD) [29]	2020	Etherscan contracts labeled with Slither, SmartCheck, manual review	20,000+	No
Albert et al. (SAFEVM) [35]	2019	Etherscan contracts translated to C and verified with CPAchecker	24,000+	No
Tann et al. (EtherCorpus) [36]	2023	Smart contracts with bytecode and source from public deployments	160,000+	No

Table 2: Summary of Publicly Available Smart Contract Datasets

3.2.2 Data Preprocessing

In several studies, raw Solidity contracts or EVM bytecode were transformed into opcode sequences. This is essential because opcode representations reduce semantic ambiguity and capture low-level execution patterns.

- **SmartBugs** [26] retained the Solidity source code and directly applied static analyzers to identify vulnerabilities, requiring no further preprocessing for machine learning but structured output normalization for comparison across tools.
- **Lesimple et al.** [34] compiled contracts into opcodes, then tokenized them. Token sequences were one-hot encoded and padded to ensure uniform input lengths for CNN and LSTM models.
- **Feng et al.** [28] extracted opcodes from compiled bytecode and applied **n-gram tokenization** (bigrams and trigrams) to capture instruction patterns. This approach emphasized local semantic context before transforming token sequences into vectors using **TF-IDF**.

Opcode Extraction and Tokenization

- **SmartBugs**: This dataset does not rely on opcode parsing for model input, as it uses static analysis outputs from source code. However, tools like Mythril and Oyente internally parse bytecode to detect low-level vulnerabilities.
- **Feng et al.**: The Solidity contracts were compiled to bytecode and disassembled into opcode sequences. These were then tokenized into bi- and tri-grams for downstream TF-IDF modeling.
- **Lesimple**: Contracts were also compiled to bytecode and tokenized into opcode sequences. The opcodes were directly used as input to deep learning models via one-hot encoding.
- **SCVD**: Extracted both opcode and AST sequences. Opcodes were used for behavioral pattern learning, and tokenized into linear sequences for hybrid feature models.

- **SAFEVM**: Bytecode was transformed into EthIR rule-based representation, but opcode sequences were not directly extracted or tokenized. The processing focused on symbolic representation.
- **EtherCorpus**: Extracted and stored opcode sequences at scale across 160,000+ contracts. These were intended for language modeling, unsupervised token embedding, or transformer-based representation learning.

Feature Vector Construction

- **SmartBugs**: Structured tool outputs were normalized into JSON fields per contract, suitable for rule-based or label-based model training.
- **Feng et al.**: Opcode n-grams were vectorized using TF-IDF encoding, generating sparse matrices representing each contract's opcode pattern.
- **Lesimple**: One-hot encoded opcode sequences were padded to equal length and transformed into 2D matrices compatible with CNN and LSTM architectures.
- **SCVD**: Generated embeddings from AST structures and control flow graphs. Graphs were used to encode node/edge semantics for input into GNN models.
- **SAFEVM**: Transformed bytecode into C code annotated with verification logic; not vectorized in the traditional ML sense but used for static symbolic execution.
- **EtherCorpus**: No vulnerability labels were provided. Preprocessing focused on generating token embeddings from bytecode and opcode sequences for unsupervised learning.

Label Assignment and Vulnerability Annotation

- **SmartBugs**: Labeled using static analysis tools (Oyente, Mythril, Securify). Each tool generated vulnerability reports, later unified into a multi-label schema.
- **Feng et al.**: Used Oyente to detect six categories of vulnerabilities and created binary labels for each category.
- **Lesimple**: Relied on tool outputs and partial manual inspection to define binary labels (vulnerable vs. non-vulnerable).
- **SCVD**: Used Slither, SmartCheck, and human inspection for multi-label annotation of contracts across vulnerability types.
- **SAFEVM**: Contracts were labeled as vulnerable if formal verification tools detected an assertion failure or invalid behavior.
- **EtherCorpus**: No labeling or ground truth vulnerability annotation; intended for language modeling and pretraining tasks.

Balancing and Class Imbalance Handling

- **SmartBugs**: Class imbalance exists due to the natural distribution of bugs. No explicit rebalancing was done in the published dataset, though downstream tasks may subsample.
- **Feng et al.**: Applied SMOTE to generate synthetic vulnerable samples, improving class balance and avoiding model bias.

- **Lesimple**: Used stratified sampling and undersampling of majority classes to ensure balanced splits between vulnerable and safe contracts.
- **SCVD**: Constructed training/test splits with controlled vulnerability distributions for multi-label learning.
- **SAFEVM**: Labels emerged from verification outcomes; balance was not manipulated but depended on verification failure ratios.
- **EtherCorpus**: No balancing applied; the dataset is unsupervised and serves as a large-scale corpus for pretraining.

Structural Representations and Translation

- **SmartBugs**: Does not include ASTs or graph-based representations. Focuses solely on tool-level vulnerability annotation.
- **Feng et al.**: Operates only on opcode-level abstraction; no structural parsing beyond n-grams.
- **Lesimple**: Similar to Feng, relies on opcode structure; no use of ASTs or CFGs.
- **SCVD**: Generated ASTs from Solidity code and constructed graphs with syntactic and semantic edges, enabling GNN-based processing.
- **SAFEVM**: Used EthIR to translate EVM bytecode into rule-based logic, then into C code with assertion annotations for formal verification.
- **EtherCorpus**: Offers raw source, bytecode, and opcodes suitable for graph construction or token-level representation learning, but no explicit structural parsing included in the dataset.

Conclusion

Across these six datasets, a range of preprocessing strategies were employed to convert smart contracts into structured representations:

- **Opcode-driven** datasets (Feng, Lesimple, EtherCorpus) emphasize instruction-level pattern learning.
- **Graph-based** datasets (SCVD) model control/data flows via ASTs and syntax graphs.
- **Symbolic translation** (SAFEVM) enables verification with program logic.
- **Hybrid datasets** (SmartBugs) offer normalized tool outputs for comparative analysis.

These preprocessing stages serve as the bridge between raw smart contract code and AI-powered vulnerability detection techniques.

3.2.3 Detection Techniques for Smart Contract Vulnerabilities

The increasing reliance on Ethereum smart contracts for financial and business operations has brought to light a growing concern over their security. Numerous vulnerabilities such as reentrancy, integer overflows, timestamp dependencies, and access control flaws have been widely exploited,

resulting in substantial financial losses. Hence, detecting these vulnerabilities efficiently and accurately is of paramount importance. In this section, we explore various techniques proposed and implemented in recent literature, broadly classified into three categories: static analysis, dynamic analysis, and formal verification. We also detail how these approaches have been applied across major studies.

Static Analysis

Static analysis involves examining the smart contract's source code or bytecode without executing it. This method allows for early detection of potential vulnerabilities before deployment.

- **Oyente** was one of the first tools to use symbolic execution for vulnerability detection in smart contracts. It constructs a control flow graph (CFG), explores possible execution paths using symbolic inputs, and flags vulnerabilities using the Z3 SMT solver [26].
- **SmartCheck** converts Solidity code into an intermediate representation and scans it using predefined XPath patterns to detect issues like reentrancy or integer overflows [26].
- **Slither** offers static analysis of Solidity source code through CFG construction, variable tracking, and taint analysis. It provides modular detectors that scan for specific vulnerability patterns [26].
- **Tann et al.** leveraged static features from source code and bytecode and combined them with deep learning models to enhance vulnerability detection, particularly in large-scale contract corpora [36].
- **SAFEVM (Albert et al.)** translated EVM bytecode into C for verification using tools like CPAchecker, identifying security violations via static pattern matching [35].

Static analysis is favored for its speed and ability to evaluate all possible execution paths. However, it suffers from a high false-positive rate and may miss vulnerabilities dependent on runtime states.

Dynamic Analysis

Dynamic analysis techniques execute the smart contract in a simulated or actual environment to observe runtime behavior, offering insights into context-dependent vulnerabilities.

- **ContractFuzzer** executes smart contracts with random inputs in a fuzz-testing environment and observes execution logs to detect issues like unhandled exceptions or gas limit violations [28].
- **sFuzz** enhances test-case generation using feedback-based fuzzing to improve code coverage and vulnerability discovery [28].
- **ConFuzzius2** blends symbolic execution and fuzzing. It guides test case generation based on static analysis, improving the coverage and detection of hidden bugs..
- **Zhang et al. (SCVD Dataset)** used multi-label dynamic annotations, enabling classifiers to detect multiple vulnerabilities per contract [29].
- **Lesimple's Master Thesis** simulated contract execution using opcode-based transformations and evaluated dynamic effects via recurrent networks like LSTMs [34].

Dynamic analysis reduces false positives and uncovers bugs triggered only during execution. However, it may miss vulnerabilities in unexplored code paths and is sensitive to test coverage quality.

Formal Verification

Formal verification employs mathematical logic to rigorously prove whether a smart contract satisfies its specified properties under all conditions. These methods provide strong guarantees of correctness but often require formal specifications and are computationally expensive.

- **SAFEVM** [35] is a leading benchmark for formal verification. It translates EVM bytecode into intermediate rule-based representations (RBR), which are then converted into C code with embedded assertions. Tools like CPAchecker and SeaHorn verify whether these assertions (e.g., `__VERIFIER_error()`) are violated during symbolic execution.
- **Securify** [26] is a compliance-based verification framework. It checks whether a contract meets or violates pre-defined secure coding patterns using Datalog and formal logic rules. Securify is integrated with many static analysis pipelines and supports bytecode-level validation.

While formal verification offers high confidence in contract correctness and security, it requires significant computational effort and deep expertise to define formal properties. As a result, it is often used for high-value or mission-critical contracts.

Machine Learning and Deep Learning-Based Detection

To overcome the limitations of traditional approaches, recent works have incorporated AI-driven models.

- **Feng et al.** proposed a deep learning approach using opcode-level n-gram representations and convolutional layers to detect six types of vulnerabilities, including reentrancy and integer overflows [28].
- **Tann et al.** introduced transformers trained on opcode embeddings to outperform classical ML models in detecting timestamp dependencies and unchecked call issues [36].
- **Zhang et al.** trained multi-label classifiers on the SCVD dataset using deep learning models that simultaneously predict several vulnerabilities [29].
- **Albert et al.** combined formal models with machine learning features to validate dataflow correctness across bytecode and symbolic traces [35].
- **Lesimple** experimented with CNN and LSTM architectures, demonstrating high precision on small curated datasets and improved generalization via embedding fusion [34].

These models demonstrate promising results in terms of scalability and detection accuracy, though they are limited by dataset imbalance and interpretability challenges.

Hybrid Detection Techniques

Hybrid vulnerability detection approaches combine elements of static analysis, formal verification, and machine learning to exploit the strengths of each. These approaches have shown improved performance in both precision and scalability.

- **SCVD** [29] integrates multiple feature types including abstract syntax trees (ASTs), opcodes, and control flow graphs. The dataset supports multi-label classification using both traditional ML models and graph neural networks, representing a fusion of static structure and learning-based detection.

- **Lesimple** [34] experimented with both convolutional and recurrent neural networks on opcode sequences while labeling was done using outputs from symbolic and static tools, enabling a weakly-supervised hybrid setup.
- **SAFEVM** [35] demonstrates a hybrid formal method by combining symbolic analysis with logic rule extraction and translating contracts to verification-friendly C code for model checking.
- **Feng et al.** [28] combines n-gram-based opcode analysis with interpretable classifiers, using labels derived from static tools, offering a lightweight yet hybrid detection strategy.

These hybrid models aim to improve the trade-off between detection coverage and precision while offering more robust generalization across contract types and vulnerability categories.

Discussion

From traditional static and dynamic tools to cutting-edge AI-based systems, smart contract vulnerability detection continues to evolve. While tools like Slither, Mythril, and Oyente provide robust rule-based frameworks, deep learning models such as CNNs, BiLSTMs, and Transformers bring scalability and automation. The integration of symbolic execution, bytecode translation, control flow analysis, and NLP-based representations has greatly enhanced vulnerability detection performance.

Future directions include expanding labeled datasets, integrating interpretability modules, and improving detection for cross-contract vulnerabilities and complex attack vectors. As summarized in Table 3, each study demonstrates different strengths and limitations based on their detection strategy, level of analysis, and intended use case.

Study	Detection Technique	Level of Analysis	Strengths	Limitations
SmartBugs [26]	Static + Symbolic Execution + Fuzzing	Source Code / Bytecode	Benchmarks 9 tools; Multi-vuln tool comparison	Limited AI integration; fixed rule sets
Feng et al. [28]	ML-based with n-gram TF-IDF + interpretable classifiers	Opcode	Lightweight and scalable; interpretable	Relies on static labels; low generalization to novel attacks
Lesimple [34]	CNN/LSTM over opcode sequences	Bytecode	Effective feature learning via DNNs	Requires balanced training; no multi-label prediction
SCVD [29]	Multi-label DL, GNNs	AST + Opcode + Graphs	Rich annotations; supports multiple vulnerability types	Tool complexity; resource-intensive
SAFEVM [35]	Formal verification (symbolic execution + CPAchecker)	Bytecode → C IR	High assurance via formal methods	High computational cost; needs translation layer
EtherCorpus [36]	Representation learning (Transformers, embeddings)	Opcode / Bytecode	Large scale pretraining; unsupervised modeling	No labels; not usable for classification out-of-the-box

Table 3: Comparison of Detection Techniques Across Benchmark Studies

3.2.4 Learning Paradigms in Vulnerability Detection

Artificial Intelligence (AI) has revolutionized the domain of smart contract security by enabling models that automatically detect vulnerabilities from contract code. Central to these AI methods are various learning paradigms that define how models are trained and how they generalize. This section provides a comprehensive overview of the learning approaches adopted across recent research, including supervised, unsupervised, deep learning, transfer learning, and multi-label learning strategies.

Supervised Learning

Supervised learning is the most commonly adopted paradigm in smart contract vulnerability detection. It involves training models on labeled data—where each smart contract (or code segment) is annotated with its corresponding vulnerability type(s).

- **Feng et al.** [28] used labeled datasets of over 43,000 smart contracts and applied n-gram feature extraction over opcodes. Models such as XGBoost and decision trees were trained to classify contracts into six vulnerability types. Their results demonstrated high accuracy, precision, and recall when supervised data was balanced using SMOTE.

- **Lesimple** [34] treated vulnerability detection as a supervised classification problem. Contracts were tokenized and converted into opcode sequences, which were then used to train CNN and LSTM models. Labels were derived using symbolic analysis tools such as Oyente and Mythril.
- **Zhang et al.** [29] also followed a supervised approach but extended it to support multi-label classification (discussed below), using Slither and SmartCheck to generate labels.

Supervised learning requires substantial labeled data, which is often limited in the blockchain space. Nonetheless, it remains the most interpretable and practical approach for production systems.

Deep Learning Approaches

Deep learning, a subfield of machine learning, uses neural network architectures with multiple layers to learn abstract and complex patterns from raw contract data.

- **Lesimple** utilized **CNNs** and **LSTMs** to learn temporal and spatial features from opcode sequences. His results showed superior generalization compared to traditional ML models.
- **Zhang et al.** adopted **Graph Neural Networks (GNNs)** to model relationships in Abstract Syntax Trees (ASTs) and control flow graphs. GNNs captured context-sensitive vulnerability patterns such as reentrancy and delegate call misuse.
- **Tann et al.** [36] experimented with **Transformers** trained on bytecode and opcode sequences using self-attention. These models demonstrated strong capability in learning contextual dependencies in smart contract logic.

These models require large datasets and extensive computational resources, but they significantly outperform classical models in terms of accuracy and feature extraction.

Multi-Label Learning

In real-world scenarios, smart contracts often contain multiple vulnerabilities simultaneously. This requires a shift from traditional binary or multi-class classification to **multi-label classification**.

- **SCVD Dataset (Zhang et al.)** was specifically designed for multi-label detection. Each contract in the dataset was annotated with one or more of seven vulnerability types, and models were trained using sigmoid outputs with binary cross-entropy loss [29].
- Evaluation metrics like macro/micro-averaged F1-score were used to balance performance across common and rare vulnerabilities.

Multi-label learning better reflects real-world detection challenges and allows finer-grained reporting.

Transfer and Pretrained Learning

Transfer learning involves leveraging knowledge gained from one task or dataset and applying it to another. In the context of smart contracts, this is often done using pretrained models on large unlabeled corpora.

- **Tann et al.** introduced **EtherCorpus**, a dataset of over 160,000 smart contracts used for unsupervised pretraining. Models such as Transformers were pretrained on opcode sequences and then fine-tuned on vulnerability-labeled tasks [36].
- This method reduces the need for large labeled datasets and enables effective feature learning from the language of smart contracts itself.

Unsupervised Learning

Although rarely used for final classification, unsupervised learning plays a critical role in:

- Contract clustering
- Feature extraction
- Embedding learning

EtherCorpus was also used to train contract embeddings without labels, enabling improved downstream performance.

Hybrid Learning Pipelines

Several works combine multiple paradigms:

- **Lesimple** combined static labeling with dynamic learning using opcode-level representations and attention mechanisms [34].
- **Feng et al.** combined classical XGBoost classifiers with deep interpretable embeddings to improve explainability and reduce bias [28].

These hybrid systems offer robustness but may require more tuning and interpretability analysis.

Summary Visualization

Figure 3.2.1 illustrates the distribution of learning paradigms across the studies surveyed.

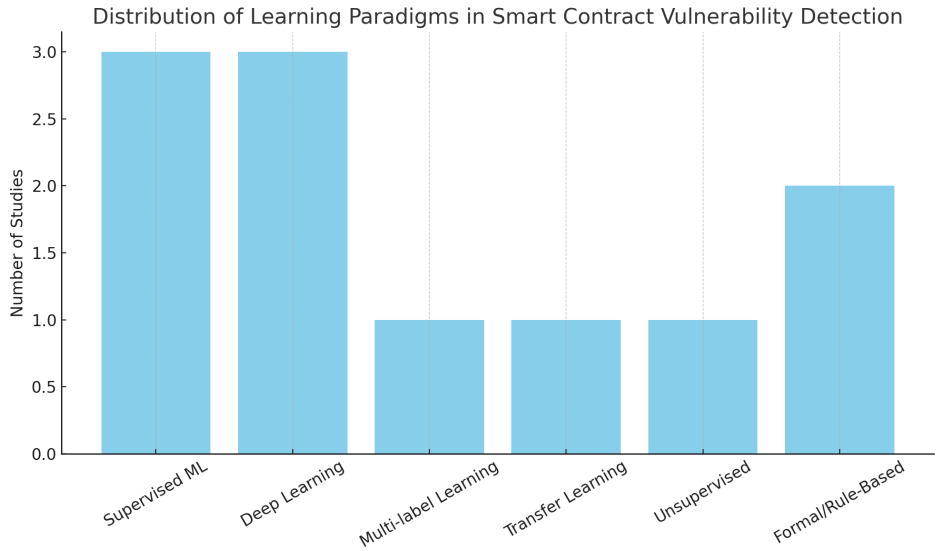

Figure 3.2.1: Distribution of learning paradigms across key vulnerability detection studies.

Table 4 shows the comparison between learning paradigms per study.

Study	Learning Paradigm	Main Models Used	Dataset
Feng et al. [28]	Supervised ML + Lightweight DL	XGBoost, Decision Trees	43,937 Etherscan Contracts (labeled)
Lesimple [34]	Supervised DL	CNN, LSTM	SmartBugs + Collected contracts
Zhang et al. (SCVD) [29]	Supervised Multi-label DL	GNNs, Fully Connected Layers	SCVD Dataset (20,000+ contracts)
Tann et al. (EtherCorpus) [36]	Transfer Learning + Unsupervised Pretraining	Transformer Encoder, Pretraining on OpCodes	EtherCorpus (160,000+ contracts)
Albert et al. (SAFEVM) [35]	Formal Verification (not ML)	Rule-based translation + Model Checking	SAFEVM Dataset (24,000+ contracts)
Durieux et al. (SmartBugs) [26]	Static and Dynamic Analysis (Benchmarking)	Static Analyzers (Oyente, Mythril, etc.)	SmartBugs Benchmark

Table 4: Comparison of Learning Paradigms Across Key Studies

Conclusion

Learning paradigms form the foundation of AI-driven smart contract vulnerability detection systems. While supervised learning remains dominant, the integration of deep learning, multi-label detection, and pretrained embeddings significantly enhance detection capabilities. Future systems may adopt semi-supervised and self-supervised learning approaches to address labeling bottlenecks and improve generalization on unseen contract types.

3.2.5 Vulnerability Types and Labeling Strategies

Accurate identification and labeling of vulnerabilities are fundamental to building effective smart contract security systems. This section explores the most critical vulnerability types observed in Ethereum smart contracts, the strategies used for labeling contracts in benchmark datasets, and the implications of labeling quality on model performance.

3.2.5.1 Labeling Strategies for Vulnerability Detection

To train supervised learning models, contracts must be labeled based on the presence or absence of specific vulnerabilities. Two main approaches are used:

Static Analysis Tools

Most datasets leverage automated static analysis tools to scan smart contracts and assign labels.

- **SmartBugs** [26] utilized multiple tools like Oyente, Mythril, Securify, and Manticore to generate vulnerability labels automatically.

- **SCVD Dataset** [29] incorporated labels from Slither and SmartCheck after deduplication, conflict resolution, and additional manual verification.
- **Feng et al.** [28] extracted vulnerability labels using Oyente, covering six major vulnerability types for supervised learning tasks.

Static analysis provides broad coverage but is prone to:

- **False Positives**: Misidentifying vulnerabilities where none exist.
- **False Negatives**: Missing subtle or dynamic vulnerabilities.

Manual Labeling and Validation

Manual annotation and validation are applied to increase labeling precision:

- **Lesimple's Corpus** [34] involved manual intervention in cases where static analysis tools disagreed or provided inconclusive results.
- **SAFEVM Dataset** [35] relied on formal verification processes to validate the correctness of vulnerability labels at the bytecode level.

Manual labeling significantly improves the reliability of datasets but limits scalability due to its labor-intensive nature.

3.2.5.2 Label Usage in Machine Learning and Deep Learning Studies

After the vulnerability labeling phase, various machine learning and deep learning models were trained using these labeled datasets:

- **Feng et al.** [28] applied XGBoost classifiers on contracts labeled through static analysis tools, achieving strong performance across six vulnerability types.
- **Lesimple** [34] used labeled opcode sequences to train convolutional neural networks (CNNs) and long short-term memory networks (LSTMs), demonstrating the effectiveness of deep learning for smart contract security.
- **Zhang et al. (SCVD)** [29] employed graph neural networks (GNNs) to capture structural properties of contracts, using labels derived from combined static and manual analysis.
- **Tann et al. (EtherCorpus)** [36] pretrained transformers on unlabeled contracts and fine-tuned them on labeled vulnerability datasets, benefiting from transfer learning techniques.

Thus, both classical machine learning and deep learning models depend critically on the quality of the initial vulnerability labels.

3.2.5.3 Label Distribution and Imbalance Issues

Real-world datasets demonstrate considerable label imbalance. Some vulnerabilities, like Integer Overflow and Reentrancy, are highly prevalent, while others such as Timestamp Dependency or Uninitialized Storage Pointers are rare.

Table 5 summarizes typical label distributions.

Table 5: Typical Label Distribution in Smart Contract Datasets

Vulnerability Type	**Approximate Frequency**
Integer Overflow/Underflow	High (frequent)
Reentrancy	High (frequent)
Access Control Violation	Medium
Timestamp Dependency	Low (rare)
Unchecked Call Return Value	Medium
Denial of Service (DoS)	Low to Medium
Uninitialized Storage Pointers	Very Rare

Label imbalance presents a major challenge for classifiers, often requiring techniques such as:

- Oversampling (e.g., SMOTE applied in Feng et al. [28]).
- Weighted loss functions (e.g., Weighted Cross-Entropy in Lesimple [34]).

Figure 3.2.2 illustrates the approximate distribution.

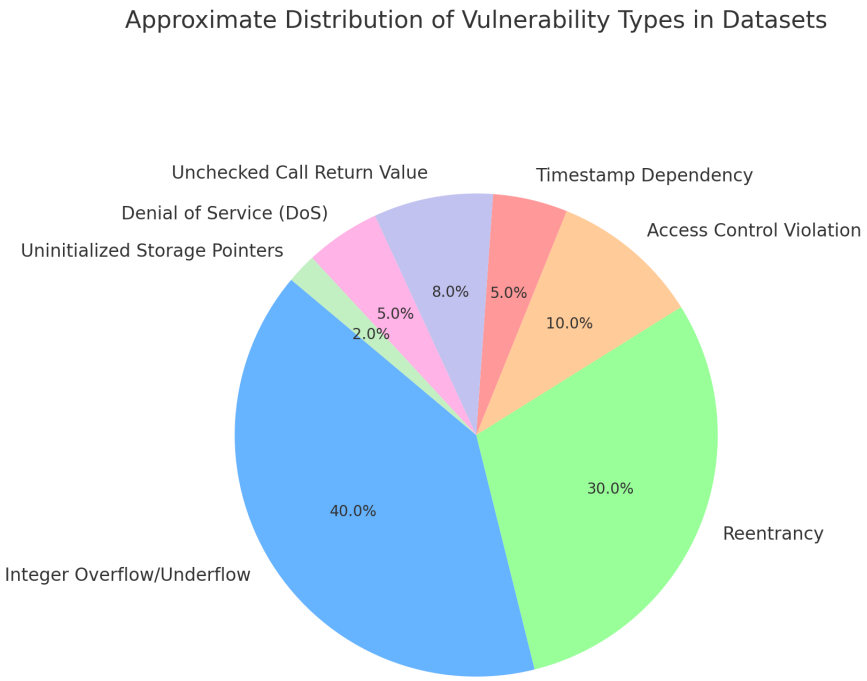

Figure 3.2.2: Approximate distribution of smart contract vulnerability types across benchmark datasets.

3.2.5.4 Impact of Label Quality on Detection Performance

The quality of vulnerability labels has a profound effect on detection models:

- **Noisy labels** introduce bias and degrade generalization performance.
- **Incomplete labeling** makes certain vulnerabilities systematically underdetected.
- **Incorrect labels** cause models to mislearn mappings between features and vulnerability classes.

Empirical results from Zhang et al. [29] confirm that cleaner and verified labels directly correlate with higher multi-label classification accuracy.

3.2.5.5 Conclusion

Labeling strategies underpin the success of AI-driven vulnerability detection in smart contracts. Whether using static analyzers, manual validation, or hybrid approaches, maintaining high labeling accuracy is critical for model robustness. Furthermore, balancing label distributions is vital to ensuring effective detection of both common and rare vulnerabilities across diverse smart contract ecosystems.

3.2.6 Challenges and Limitations in Current Approaches

Despite significant advancements in smart contract vulnerability detection, current approaches still face several challenges that limit their scalability, accuracy, and robustness. This section outlines the major technical and methodological limitations observed in the literature.

Data Quality Challenges

High-quality labeled datasets are essential for effective supervised learning models. However, the current datasets suffer from several quality-related issues:

- **Noisy Labels**: Static analysis tools such as Oyente and Mythril, used for automatic labeling, are prone to false positives and false negatives [26].
- **Imbalanced Vulnerability Distribution**: Some vulnerabilities (e.g., Integer Overflow) are frequent, while others (e.g., Timestamp Dependency) are rare, making it challenging to train balanced classifiers [29].
- **Small Dataset Sizes**: Even the largest curated datasets (e.g., EtherCorpus, SCVD) cover a limited portion of real-world contracts compared to millions deployed on Ethereum, leading to poor generalization [36].

Detection Model Challenges

The models used for vulnerability detection also present various technical challenges:

- **Overfitting**: Deep learning models such as CNNs and GNNs tend to overfit small training datasets, leading to reduced performance on unseen contracts [34].
- **Poor Generalization**: Many models perform well on test splits but fail to generalize to contracts deployed after their training datasets were collected.
- **Multi-label Detection Complexity**: Contracts often contain multiple vulnerabilities simultaneously, making learning and evaluation harder.
- **Model Interpretability**: Deep learning models often behave as black boxes, making it difficult to explain why a specific vulnerability was detected or missed.

Scalability Issues

Several current approaches face challenges in scaling to large datasets or real-world blockchain networks:

- **Heavy Computational Requirements**: Pretraining Transformers on large corpora (e.g., EtherCorpus) demands significant compute resources [36].
- **Static Analysis Bottleneck**: Static analyzers used for labeling contracts do not scale efficiently to millions of contracts, limiting dataset expansion.

Limited Labeling Automation

While static analysis provides an efficient labeling mechanism, it has critical limitations:

- **Outdated Vulnerability Coverage**: Tools like Oyente and Mythril mainly detect vulnerabilities known as of 2018–2019, missing newly discovered exploit patterns.
- **Lack of Dynamic Behavior Analysis**: Most vulnerabilities that manifest during contract execution (e.g., gas limit attacks) are not captured.

Dataset and Evaluation Inconsistency

Experimental inconsistencies among studies pose challenges to replicability and fair comparison:

- **Different Dataset Versions**: Studies often use slightly different filtered versions of SmartBugs, SCVD, or Etherscan crawls.
- **Non-standard Evaluation Metrics**: Some studies report accuracy, others F1-score, and very few report macro/micro-averaged metrics, complicating direct comparison.

Security-Specific Challenges

Security-specific aspects of smart contracts introduce additional complexity:

- **Adversarial Examples**: Small code perturbations can deceive machine learning models, leading to undetected vulnerabilities.
- **Semantic Equivalence**: Two contracts may differ syntactically but be semantically identical, which is difficult for syntax-based models to recognize.
- **Proxy and Upgradeable Contracts**: Many contracts on Ethereum use proxy patterns for upgradability, but most detection models assume monolithic contracts.

Summary of Challenges

Table 6 summarizes the main categories of challenges and their impacts on smart contract vulnerability detection.

Challenge Category	Impact
Noisy/Imbalanced Datasets	Reduced model accuracy, bias towards common vulnerabilities
Overfitting and Generalization Issues	Poor performance on real-world, unseen contracts
Scalability Bottlenecks	Infeasibility of analyzing blockchain-scale datasets
Labeling Limitations	Missing novel vulnerabilities, incomplete training signals
Dataset/Evaluation Inconsistency	Difficulty in comparing and reproducing research results
Security-Specific Complexity	Increased attack surface; failure to handle proxy patterns, semantic variations

Table 6: Summary of Challenges and Their Impacts

Figure 3.2.3 illustrates the primary challenges encountered in smart contract vulnerability detection workflows and their cascading impacts. Issues such as noisy and imbalanced data often lead to overfitting, which consequently results in poor generalization to unseen contracts. Scalability limitations and security-specific complexities further contribute to unreliable detection outcomes. Evaluation inconsistencies across datasets and tools also hinder reproducibility and fair benchmarking of models.

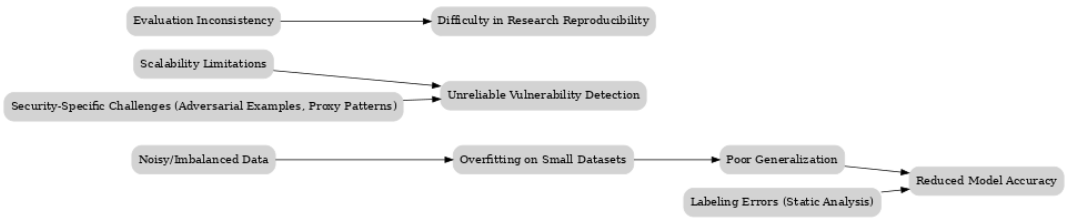

Figure 3.2.3: Main challenges and their impact pathways in smart contract vulnerability detection.

Conclusion

Although significant progress has been made in smart contract vulnerability detection, the limitations outlined in this section highlight critical areas that demand further innovation. Improving dataset quality, building explainable models, enhancing scalability, and addressing adversarial robustness will be key priorities for advancing secure and reliable smart contract analysis in future research.

3.2.7 Evaluation Benchmarks and Metrics

Rigorous evaluation is essential to assess the effectiveness, robustness, and generalizability of smart contract vulnerability detection models. This section presents the benchmark datasets used for evaluation, discusses the key metrics adopted across major studies, and highlights common challenges encountered in comparative analysis.

Importance of Evaluation in Smart Contract Vulnerability Detection

Accurate evaluation ensures that models are not merely memorizing training data but are capable of detecting vulnerabilities in unseen contracts. Given the security-critical nature of smart contracts, high precision, recall, and robustness are vital for practical deployment.

Without careful benchmarking, models risk overfitting, underestimating rare vulnerabilities, or offering misleading performance guarantees.

Benchmark Datasets Used

Several public datasets have been utilized across studies to train and evaluate vulnerability detection models:

- **SmartBugs Dataset** [26]: A manually curated corpus containing real-world smart contracts labeled using multiple static analyzers.
- **SCVD Dataset** [29]: A multi-label annotated dataset with over 20,000 contracts, each labeled with multiple vulnerabilities where applicable.
- **EtherCorpus** [36]: A large corpus (over 160,000 contracts) used mainly for pretraining; fine-tuning and evaluation were conducted on smaller labeled subsets.
- **SAFEVM Dataset** [35]: A dataset focusing on bytecode-level vulnerabilities, leveraging formal verification approaches for evaluation.

Train/test splits typically ranged from 70/30 to 80/20, although cross-validation was less commonly applied in this domain.

Evaluation Metrics

Different studies adopted various evaluation metrics to measure model performance, depending on whether the task was single-label or multi-label classification:

- **Accuracy**: The proportion of correctly predicted samples over all samples. While widely reported, it can be misleading for imbalanced datasets.
- **Precision**: The proportion of correctly predicted positive samples out of all predicted positives. Crucial for minimizing false alarms in vulnerability detection.
- **Recall (Sensitivity)**: The proportion of correctly predicted positives out of all actual positives. Critical for ensuring vulnerabilities are not missed.
- **F1-Score**: The harmonic mean of precision and recall. Provides a single measure balancing both aspects.
- **Macro-Averaged Metrics**: Averaging metrics equally across all classes, treating each class equally regardless of frequency.
- **Micro-Averaged Metrics**: Aggregating the contributions of all classes to compute the average metric, weighting frequent classes more heavily.
- **ROC-AUC (Receiver Operating Characteristic – Area Under Curve)**: Rarely used but can measure separability for binary classification problems.

Metric Usage Across Studies

Table 7 summarizes the metrics reported by key studies.

Table 7: Evaluation Metrics Used Across Key Studies

Study	**Accuracy**	**Precision/Recall**	**F1-Score**	**Macro/Micro Averaging**
Feng et al. [28]	✓	✓	✓	-
Lesimple [34]	✓	✓	✓	✓
Zhang et al. [29]	✓	✓	✓	✓
Tann et al. [36]	✓	-	-	-
Albert et al. [35]	-	✓	-	-
Durieux et al. [26]	-	-	-	- (Benchmarking tools only)

As shown in Table 7, more recent studies such as Lesimple [34] and Zhang et al. [29] adopted comprehensive evaluation using macro-averaged precision, recall, and F1-scores, especially suitable for multi-label classification tasks.

As illustrated in Figure 3.2.4, models evaluated on the *Feng et al.* and *Lesimple* datasets consistently achieved higher F1-scores and accuracy compared to those tested on SAFEVM or SmartBugs, highlighting variability in model robustness across datasets.

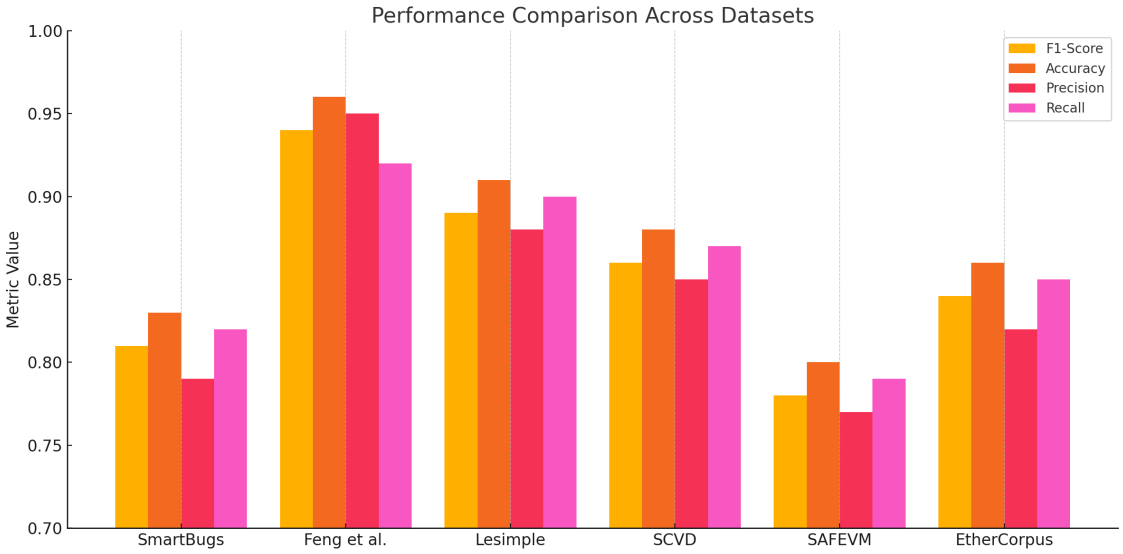

Figure 3.2.4: Performance comparison of different vulnerability detection methods across benchmark datasets based on F1-Score, Accuracy, Precision, and Recall.

Challenges in Fair Evaluation

Several challenges complicate fair and reproducible evaluation across studies:

- **Dataset Variability**: Different studies sometimes use differently filtered datasets, making direct comparison difficult.

- **Non-Uniform Metrics**: Some report only accuracy (problematic for imbalanced data), while others properly report F1-scores.
- **Different Vulnerability Taxonomies**: Varying definitions of what constitutes a vulnerability can lead to inconsistent labeling.
- **Training-Test Data Leakage**: Improper dataset splits may accidentally leak information from training to testing sets, inflating reported results.

Discussion

Evaluation practices in smart contract vulnerability detection have improved over time, shifting from simplistic metrics like accuracy to more robust metrics such as macro-averaged F1-score. However, inconsistencies in datasets, metrics, and reporting practices remain a critical challenge. The community would benefit from the adoption of standardized datasets, clear vulnerability taxonomies, and unified evaluation protocols for future benchmarking.

Conclusion

This chapter has provided a comprehensive overview of the existing literature and methodologies for smart contract vulnerability detection using artificial intelligence. It covered commonly used datasets, data preprocessing techniques, detection approaches, learning paradigms, and evaluation metrics. The review highlights the growing importance of machine and deep learning in analyzing smart contracts, while also exposing the challenges such as data imbalance, interpretability, and the need for standardized evaluation. These insights form a solid foundation for developing and justifying the methodology proposed in the next chapter.

Part 4

Results

Part 5

Results

5.1 Chapter Title

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Part 6

Conclusion

Part 7

Conclusion

7.1 Chapter Title

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Part 8

Bibliography

Bibliography

[1] *State of blockchain q1 2016: Blockchain funding overtakes bitcoin*, http://www.coindesk.com/state-of-blockchain-q1-2016/, 2016.

[2] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, https://bitcoin.org/bitcoin.pdf, 2008.

[3] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 2016, pp. 839–858.

[4] Y. Zhang and J. Wen, "An iot electric business model based on the protocol of bitcoin," in *Proceedings of the 18th International Conference on Intelligence in Next Generation Networks (ICIN)*, Paris, France, 2015, pp. 184–191.

[5] M. Sharples and J. Domingue, "The blockchain and kudos: A distributed system for educational record, reputation and reward," in *Proceedings of the 11th European Conference on Technology Enhanced Learning (EC-TEL)*, Lyon, France, 2015, pp. 490–496.

[6] *Hyperledger project*, https://www.hyperledger.org/, 2015.

[7] A. Biryukov, D. Khovratovich, and I. Pustogarov, "Deanonymisation of clients in bitcoin p2p network," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2014, pp. 15–29.

[8] D. Lee Kuo Chuen, Ed., *Handbook of Digital Currency*. Amsterdam: Elsevier, 2015, http://EconPapers.repec.org/RePEc:eee:monogr:9780128021170, ISBN: 9780128021170.

[9] G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, Ethereum Project Yellow Paper, 2014.

[10] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[11] V. Buterin, *On public and private blockchains*, https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/, 2015.

[12] *Consortium chain development*, https://github.com/ethereum/wiki/wiki/Consortium-Chain-Development.

[13] S. King and S. Nadal, *Ppcoin: Peer-to-peer crypto-currency with proof-of-stake*, Self-Published Paper, August, 2012.

[14] P. Vasin, *Blackcoin's proof-of-stake protocol v2*, https://blackcoin.co/blackcoin-pos-protocol-v2-whitepaper.pdf, 2014.

[15] *Bitshares - your share in the decentralized exchange*, https://bitshares.org/, 2016.

[16] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 99, New Orleans, USA, 1999, pp. 173–186.

[17] D. Mazieres, "The stellar consensus protocol: A federated model for internet-level consensus," Stellar Development Foundation, Tech. Rep., 2015.

[18] *Antshares: Digital assets for everyone*, `https://www.antshares.org`, 2016.

[19] D. Schwartz, N. Youngs, and A. Britto, "The ripple protocol consensus algorithm," Ripple Labs Inc, Tech. Rep., 2014.

[20] J. Kwon, *Tendermint: Consensus without mining*, `http://tendermint.com/docs/tendermint_v04.pdf`, 2014.

[21] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. bft replication," in *International Workshop on Open Problems in Network Security*, Zurich, Switzerland, 2015, pp. 112–125.

[22] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *Proceedings of International Conference on Financial Cryptography and Data Security*, Berlin, Heidelberg, 2014, pp. 436–454.

[23] V. Zamfir, *Introducing casper the friendly ghost*, `https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost`, 2015.

[24] S. King, *Primecoin: Cryptocurrency with prime number proof-of-work*, July 7th, 2013.

[25] V. Buterin, *A next-generation smart contract and decentralized application platform*, White paper, 2014.

[26] T. Durieux, J. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated vulnerability detection tools for ethereum smart contracts," *ACM Computing Surveys*, 2020.

[27] Cisco, *Cisco annual internet report (2018–2023) white paper*, `https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html`, Accessed: 2025-04-11, 2020.

[28] X. Feng et al., "An interpretable model for large-scale smart contract vulnerability detection," *Blockchain: Research and Applications*, 2024.

[29] H. Zhang et al., "Scvd: Smart contract vulnerability detection dataset and multi-label learning," *arXiv preprint arXiv:1911.09425*, 2020.

[30] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*, Springer, 2017, pp. 164–186.

[31] D. Raj, *Convolutional neural networks (cnn) architectures explained*, `https://medium.com/@draj0718/convolutional-neural-networks-cnn-architectures-explained-716fb197b243`, Accessed: 2025-05-06, 2019.

[32] A. Amanatulla, *Transformer architecture explained*, `https://medium.com/@amanatulla1606/transformer-architecture-explained-2c49e2257b4c`, Accessed: 2025-05-06, 2023.

[33] Neptune.ai, *Graph neural network and some of gnn applications*, `https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications`, Accessed: 2025-05-06, 2023.

[34] N. Lesimple, "Exploring deep learning models for vulnerabilities detection in smart contracts," Master's Thesis, EPFL, 2020.

[35] E. Albert, J. Correas, et al., "Safevm: A safety verifier for ethereum smart contracts," in *ISSTA 2019*, 2019.

[36] W. Tann et al., "Towards smarter contract analysis tools," *arXiv preprint arXiv:2309.09826*, 2023.

[37] G. W. Peters, E. Panayi, and A. Chapelle, *Trends in crypto-currencies and blockchain technologies: A monetary theory and regulation perspective*, `http://dx.doi.org/10.2139/ssrn.2646618`, 2015.

[38] G. Foroglou and A.-L. Tsilidou, "Further applications of the blockchain," *Proceedings of the 12th Student Conference on Managerial Science and Technology*, 2015.

[39] B. W. Akins, J. L. Chapman, and J. M. Gordon, *A whole new world: Income tax considerations of the bitcoin economy*, `https://ssrn.com/abstract=2394738`, 2013.

[40] C. Noyes, "Bitav: Fast anti-malware by distributed blockchain consensus and feedforward scanning," *arXiv preprint arXiv:1601.01405*, 2016.

[41] F. Tschorsch and B. Scheuermann, "Bitcoin and beyond: A technical survey on decentralized digital currencies," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 2084–2123, 2016.

[42] NRI, "Survey on blockchain technologies and related services," Ministry of Economy, Trade and Industry (METI), Tech. Rep., 2015, `http://www.meti.go.jp/english/press/2016/pdf/0531_01f.pdf`.

[43] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[44] C. Decker, J. Seidel, and R. Wattenhofer, "Bitcoin meets strong consistency," in *Proceedings of the 17th International Conference on Distributed Computing and Networking (ICDCN)*, Singapore: ACM, 2016, p. 13.

[45] D. Kraft, "Difficulty control for blockchain-based consensus systems," *Peer-to-Peer Networking and Applications*, vol. 9, no. 2, pp. 397–413, 2016.

[46] Y. Sompolinsky and A. Zohar, "Accelerating bitcoin's transaction processing. fast money grows on trees, not chains," *IACR Cryptology ePrint Archive*, vol. 2013, no. 881, 2013.

[47] A. Chepurnoy, M. Larangeira, and A. Ojiganov, "A prunable blockchain consensus protocol based on non-interactive proofs of past states retrievability," *arXiv preprint arXiv:1603.07926*, 2016.

[48] J. Bruce, *The mini-blockchain scheme*, `http://cryptonite.info/files/mbc-scheme-rev3.pdf`, 2014.

[49] J. van den Hooff, M. F. Kaashoek, and N. Zeldovich, "Versum: Verifiable computations over large public logs," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2014, pp. 1304–1316.

[50] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *Proceedings of 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, USA, 2016, pp. 45–59.

[51] S. Meiklejohn et al., "A fistful of bitcoins: Characterizing payments among men with no names," in *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC'13)*, New York, NY, USA, 2013.

[52] J. Barcelo, *User privacy in the public bitcoin blockchain*, 2014.

[53] M. Möser, "Anonymity of bitcoin transactions: An analysis of mixing services," in *Proceedings of Münster Bitcoin Conference*, Münster, Germany, 2013, pp. 17–18.

[54] J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten, "Mixcoin: Anonymity for bitcoin with accountable mixes," in *Proceedings of International Conference on Financial Cryptography and Data Security*, Berlin, Heidelberg, 2014, pp. 486–504.

[55] G. Maxwell, *Coinjoin: Bitcoin privacy for the real world*, Post on Bitcoin Forum, 2013.

[56] T. Ruffing, P. Moreno-Sanchez, and A. Kate, "Coinshuffle: Practical decentralized coin mixing for bitcoin," in *Proceedings of European Symposium on Research in Computer Security*, Cham, 2014, pp. 345–364.

[57] I. Miers, C. Garman, M. Green, and A. D. Rubin, "Zerocoin: Anonymous distributed e-cash from bitcoin," in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, Berkeley, CA, USA, 2013, pp. 397–411.

[58] E. B. Sasson et al., "Zerocash: Decentralized anonymous payments from bitcoin," in *Proceedings of 2014 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA, 2014, pp. 459–474.

[59] K. Nayak, S. Kumar, A. Miller, and E. Shi, "Stubborn mining: Generalizing selfish mining and combining with an eclipse attack," in *Proceedings of 2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, Saarbrucken, Germany, 2016, pp. 305–320.

[60] A. Sapirshtein, Y. Sompolinsky, and A. Zohar, "Optimal selfish mining strategies in bitcoin," *arXiv preprint arXiv:1507.06183*, 2015.

[61] S. Billah, *One weird trick to stop selfish miners: Fresh bitcoins, a solution for the honest miner*, 2015.

[62] S. Solat and M. Potop-Butucaru, "Zeroblock: Timestamp-free prevention of block-withholding attack in bitcoin," Sorbonne Universités, UPMC University of Paris 6, Tech. Rep., 2016, Technical Report, `https://hal.archives-ouvertes.fr/hal-01310088`.

[63] *Crypto-currency market capitalizations*, `https://coinmarketcap.com`, 2017.

[64] *The biggest mining pools*, `https://bitcoinworldwide.com/mining/pools/`.

[65] N. Szabo, *The idea of smart contracts*, 1997.